Towards Solving Essence With Local Search: a Proof of Concept Using Sets and Multisets

Saad Attieh, Christopher Jefferson, Ian Miguel, and Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, UK
{sa74,caj21,ijm,pwn1}@st-andrews.ac.uk

Abstract. We propose a local search solver that operates directly on the high level structures found in the Essence abstract constraint specification language. High quality neighbourhoods are automatically derived from the structured variable types such as set, multiset, set of sets etc. The solver we present is distinguished from other local search solvers as it can operate directly on the high level types in Essence without refining such types into low level representations. This provides a major scalability advantage for problems with nested structures such as set of set, since the solver dynamically adds and deletes constraints as the sizes of these structures vary during search. The Essence language contains many abstract variable types. In this paper, we present an implementation that supports multi sets and sets as a proof of concept. We outline the framework required to perform local search on Essence expressions, covering incremental evaluation, dynamic unrolling and neighbourhood construction. The solver is benchmarked against other constraint programming and local search solvers on three problem classes: Sonet, The Knapsack Problem, and The Golomb Ruler Problem. Future work will focus on broadening the range of types supported by the local solver.

1 Introduction

Constraint modelling languages, such as MiniZinc [17] or Essence [9,10,11] offer to users a convenient means of expressing a constraint problem without concerning themselves with the specific details of a particular constraint solver. We focus herein on the Essence language, which is characterised by its support for abstract type such as set, multiset, function and partition, and particularly by its support for nesting of these types, such as set of multisets, or multiset of functions. To illustrate, consider the Essence specification of the Synchronous Optical Networking Problem (Sonet, problem 56 at www.csplib.org. See also [12,25]) in Figure 1. An Essence specification identifies: the input parameters of the problem class (`given`), whose values define an instance; the combinatorial objects to be found (`find`); the constraints the objects must satisfy (`such that`); identifiers declared (`letting`); and an (optional) objective function (`min/maximising`). In this example, the single abstract decision variable `network` is a multiset of sets, representing the rings on which communicating nodes are installed.

```
given nnodes, nrings, capacity : int(1..)
letting Nodes be domain int(1..nnodes)
$ connections that must be achieved between the nodes
given demand : set of set (size 2) of Nodes

find network :
  mset (size nrings) of set (maxSize capacity) of Nodes

such that
$All connections between nodes are achieved
forAll pair in demand .
    exists ring in network .
        pair subsetEq ring

$ objective: minimise total number of connections to rings.
$ i.e. minimise sum of the size of each ring.
minimising sum ring in network . |ring|
```

Fig. 1: Essence specification of the Synchronous Optical Networking problem. A set of rings is used to facilitate to communication between nodes. A node may be installed onto multiple rings. The task is to ensure that all pairs of nodes that require to communicate, given in `demand`, are able to do so while minimising the total number of installations onto the rings.

The Conjure automated constraint modelling system [1,2,4,5] refines an Essence specification into a solver-independent constraint model in the Essence Prime modelling language [19], where the abstract decision variables are represented as constrained collections of primitive variables, such as integer or Boolean variables. The Savile Row system [18,19,20] then transforms and prepares the Essence Prime model for input to a particular constraint solver, such as Minion [13], or SAT.

Refinement obscures the abstract structure apparent in the original Essence specification, which is a particular problem for constructing neighbourhoods for local search. In recent work [3], we presented Structured Neighbourhood Search (SNS), a method for generating neighbourhoods in Essence (i.e. pre-refinement) and then refining them along with the problem specification. In this paper, we describe a proof-of-concept solver Athanor that takes an alternative approach: operating directly on an Essence specification, performing local search on the abstract variables. We motivate this approach below.

1.1 Solving Essence Directly

Our hypothesis is that there are two principal benefits of performing local search directly over the high level variables in an Essence specification: the guidance provided by the structure apparent in the abstract types in Essence, and scalability. Both advantages hold over approaches that generate neighbourhoods starting from a lower level representation. The scalability advantage holds over the SNS approach of refining neighbourhoods generated from an Essence specification.

First, the type information in the Essence specification is a significant advantage in generating effective neighbourhoods for local search automatically. In the Sonet example, it is clear directly from the specification that the problem requires us to find a multiset of sets of nodes. Hence, neighbourhoods related to sets and multisets, such as to add to or remove from a (multi)set or to exchange elements between sets (i.e. neighbourhoods that preserve the multiset/set structure), can be generated straightforwardly. By contrast, an equivalent constraint model in MiniZinc [17] or Essence Prime [19] must represent this abstract decision variable with a constrained collection of more primitive variables, such as the matrix models [7,8] presented by Smith [25]. In this context, it is significantly more difficult to recognise the structure (i.e. the multiset of sets) in the problem and generate the equivalent neighbourhoods.

Second, there is a great scalability advantage due to the fact that the values of the abstract types in Essence can vary significantly in size. As a simple illustrative example, consider the Essence type `set of int(1..n)`. A natural model is to employ n Boolean variables indicating which of the n integers are in the set. However, all values of the set domain that participate in solutions may have much smaller cardinality. Furthermore, a quantified expression requiring a constraint to be posted per element of the set will result in n (guarded) constraints to support the possibility of all of the integers $1 \dots n$ being in the set. In contrast, a solver that natively understands Essence types can support a dynamically sized set where values and constraints are introduced or deleted as the cardinality of the set changes during search. For nested types, such as set of set of int or multiset of partition, the difference in size between the minimum and maximum value of the domain of the Essence variable (and hence its representation in a constraint model) can be dramatic.

1.2 Overview

This paper presents the framework that has been constructed to support solving Essence specifications directly in the Athanor prototype. Though we make mention of several Essence types, Athanor currently supports only set and multiset type constructors, which can be arbitrarily nested. The same methodology that is described in the following sections will be applied to the rest of the Essence types in future work. Following a discussion of the overall solver architecture in Section 2, Section 3 describes the implementation of an incremental evaluator for constraints. Section 4 briefly discusses constraint violations and how they have been integrated with high level structured types. Section 5 describes the method of dynamically adding and removing constraints as items are added and removed from set variables. After the search process is outlined in Section 6, three case studies on Sonet, the Knapsack Problem, and the Golomb

Ruler Problem are presented to demonstrate the operation of Athanor in practice and test our research hypotheses.

2 Solver Architecture

Many constraint solvers perform a complete search of the search space and hence are able to prove optimality for constrained optimisation problems, or the existence or absence of a solution for satisfaction problems. Local search solvers sacrifice completeness in order to focus on finding good quality solutions quickly. They employ a selection of *heuristics* (moves) that are iteratively applied to an assignment to the decision variables (the *active solution*) with the aim of finding a better assignment. Given a set of heuristics or moves, a *metaheuristic* is used to select the heuristic to apply at each step of the search. A metaheuristic is a search procedure such as Hill Climbing [24, Chapter 4], Simulated Annealing [16] or Tabu Search [14]. A *neighbourhood* is the search space that can be explored by a given move. In this paper, we present a method of automatically deducing good quality neighbourhoods from the abstract types in the Essence language. We show how even a simple search procedure (Hill Climbing) is able to achieve high-quality solutions quickly using the automatically constructed neighbourhoods.

There are other local search methods that are able to generate neighbourhoods from the set of constraints and decision variables, however they operate on a low-level description of a problem instance where the nested structure of Essence types (for example a set of sets) has already been lost. For example there are two Large Neighbourhood Search (LNS) methods, Propagation-Guided LNS [21] and Explanation-Based LNS [23] that are able to generate neighbourhoods dynamically from the constraints and the state of search. Constraint-Based Local Search (CBLS) methods such as OscaR/CBLS [6] can also derive neighbourhoods automatically from the set of constraints and variables. We compare with all three in our experiments.

In our earlier work on Structured Neighbourhood Search [3] we generated neighbourhoods from Essence specifications (as we do in this paper) then applied Conjure and Savile Row to refine them (alongside the model) into the input language of a backtracking constraint solver. We used an adapted version of the Minion solver that applies the neighbourhoods in a similar way to LNS. This approach contrasts with the work described in this paper where the neighbourhoods are applied directly to the Essence variables.

3 Incremental Evaluation

In order to decide whether or not to accept a move, the search procedure must be able to evaluate whether or not the move has resulted in a solution better than the active solution. The process by which this is done is described in this and the following sections. If the search procedure accepts the change, the new set of assignments becomes the active solution. Otherwise, the active solution is left unchanged and a new move is selected. To evaluate a move efficiently, we must update the state of the solver in an incremental way, avoiding the need to recompute the entire state of the solver.

The solver 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 assigned to a particular value from their respective domains. Hence the constraint AST can be evaluated to a single Boolean value and the objective AST can be evaluated to a single integer. However, the ASTs are maintained so that they support incremental evaluation. Incremental evaluation takes advantage of the fact that if a variable is reassigned to a new value only its ancestors, the nodes on the path from the associated leaf back to the root, must be reevaluated.

At the start of search, the solver begins by assigning a random value to each of the variables and performing a full evaluation of the AST. Afterwords, in order to facilitate incremental evaluation, every node in the AST attaches a trigger to each of its children, a call back which is invoked notifying the parent of changes to the child nodes that might affect the value yielded by the parent. Every type of node (integer returning, set returning, etc.) can trigger at least two types of events:

- `possibleValueChange()`, notifying the parent that the value yielded by the child may change. It allows the parent to record any properties of the child before its value is altered. This event must precede any change to the child but it is legal for no change to actually take place.
- `valueChanged()`, notifying the parent that the value yielded by the child has changed.

However, for higher level types such as set and multi set, simply indicating that a value has changed can greatly hinder incremental evaluation since these types are composed of many elements. Such an event gives no indication as to how many of the elements in the set need reevaluating. Therefore, high level types can make use of more descriptive events. For example, a set also has:

- `valueAdded()`,
- `valueRemoved()`,
- `possibleMemberValueChange()`,
- `memberValueChange()`.

Of course, `memberValueChange()` can be formed by composing `valueRemoved()` and `valueAdded()` but as shown in the next section, it can be useful to consider these two events as one. Also note that a constraint may choose to attach triggers to a set as a whole or trigger on the items within.

Figure 2 gives an example AST state during incremental evaluation, using the expression describing the objective of the Sonet problem in Figure 1.

The process that takes place when adding the number 3 to the set `ring2` is as follows:

```
sum ring in network .
        |ring|
```

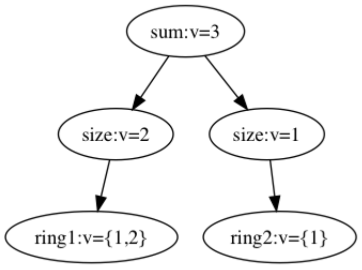

Fig. 2: Incremental evaluation. $|x|$ means size of x

- `ring2` triggers the event `possibleValueChange()` and this is echoed all the way to the root.
- When `sum` receives the `possibleValueChange()` event, it caches the value of the operand that triggered the event, `size:v=1`.
- The integer 3 is then added to `ring2`. Its value is now `ring2:v={1,3}`,
- the event `valueAdded()` is sent to the parent.
- `size` updates its value to `size:v=2`, the event `valueChanged()` is sent to its parent.
- `sum` updates its value by subtracting the old value `size:v=1` and adds the new value `size:v=2`. The old value was cached when the `sum` received the `possibleValueChange()` event at the start.
- The node now has the value `sum:v=4`, the event `valueChanged()` is forwarded to its parent...

4 Violation counts

In Figure 2 we showed how the values of integers and sets are incrementally updated. A similar procedure is used for Boolean expressions, which have been extended to store a violation count. A violation count is an integer, a heuristic that gives an indication as to the magnitude of the change necessary to the set of assignments such that the constraint is satisfied. Our methods of calculating violations are inspired by van Hentenryck and Michel [15]. For example, given two integers x and y and the constraint $c(x = y)$, the violation on c $v(c) = |x-y|$. A violation count is also attributed to the variables in the scope of a constraint. These variable violation counts are a heuristic used to give an indication as to what extent each variable contributes towards the violating constraints. The violation count on a variable u is the sum of violations attributed to u by the constraints posted on u. This helps to guide the solver when selecting which variables to modify when searching for a feasible solution. However, previous work on violation counts have only made reference to variables with integer or boolean types. We must examine how to extend the method to variables with nested types. Consider the example shown in Figure 3 in which a constraint is posted on to the integers contained within a set.

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

Fig. 3: Constraint on a nested type, $c(i)$ is some arbitrary constraint on i

We must consider how violations are attributed when elements in s violate the constraint c. As mentioned, the violation counts on variables are supposed to guide the solver towards identifying the cause of violating constraints. Hence, the rules for attributing violations to nested types are as follows:

– When a violation is attributed to an element i of a 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 outmost structure.
– However, if i is itself a containing structure, the violation is not attributed to any of the elements in i. Neither is the violation propagated to any of the siblings of i, that is, other elements contained in s.
– Hence, the violation on any containing structure s is the sum of violations attributed to s plus the sum of violations attributed to the elements in s.

Consider the constraint $|s| = 1$ (the size of s is 1). If this constraint is violated, all the elements in s are all equally to blame. Therefore, there is no benefit in attributing the violation to the elements in s, rather s as a whole is assigned a violation. However, if the constraint is like that shown in Figure 3, it makes sense to assign a violation to only those elements in s that are causing the violation, so that the solver may be biased towards altering their value. 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 an Essence specification has a fixed number of abstract variables, these variables are usually of container types (set, sequence, multi set, and so on). The values of such variables can vary considerably in size and hence new elements can be introduced or deleted during search. Essence also allows the quantification over such containers attaching a constraint to each of the elements within. Therefore, it must be possible to add and delete constraints in accordance with the changes in size of the values of the variables being quantified over.

Consider Figure 4, for example. In the AST representation, s is an operand of the `forAll` node. The `forAll` node also has one operand for each item in s. The `forAll` node also stores the expression (`i%2=0`) that is to be applied to each element in the set. This expression is a template, meaning that it is represented by an incomplete AST. The AST is incomplete as i in the expression does not refer to one variable instance. Rather, it is used to refer to each

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

Fig. 4: Quantifying over a set

value in the set. We call this the iterator. When s changes from being empty to having one element, an operand is added to the `forAll` node, the expression template is copied in and made complete by assigning the iterator to the newly added element. The AST subtree representing the copied expression is than fully evaluated much like the evaluation of the entire AST at the start of search. The nodes in the subtree then begin triggering on their children as per Section 3.

However, as more elements are added, rather than copying the unevaluated expression template, the expression subtree most recently added to the `forAll` node is copied. This is because the expression would have already been evaluated. All that is necessary is to assign the iterator to point to a new element and a `valueChanged()` event passed up the subtree. As might be expected, as elements are deleted from the set, their corresponding subtrees are also removed.

6 Search

As demonstrated in the following case studies, the solver uses the variable types present in the problem to derive a set of neighbourhoods: actions that may be performed on variables in order to improve either the violation count or the objective. A simple search strategy is built upon the set of derived neighbourhoods:

```
procedure ATHANOR
    $\sigma \leftarrow$ available neighbourhoods
    Assign all variables to random value REPAIR    ▷ Move to feasible solution
    $w \leftarrow 1$    ▷ Used in calls to explore, controls violation allowed
    while time limit not reached do
        CLIMB
        $w \leftarrow 1$
        repeat
            $w \leftarrow$ CEILING$(1.5w)$
            EXPLORE(w)
            REPAIR
        until better feasible solution
    end while
end procedure
```

```
procedure EXPLORE(w)
    $o \leftarrow$ current objective
    $v \leftarrow$ total violation
    $i \leftarrow 0$                                  ▷ iterations spent without improving
    $\sigma \leftarrow$ available neighbourhoods
    while $i \leq$ limit do                           ▷ limit is a tunable parameter
        select a random neighbourhood $n \in \sigma$
        $(\mathrm{o2}, \mathrm{v2}) \leftarrow$ RUNNEIGHBOURHOOD($n$)
        if $\mathrm{v2} \leq w \wedge \mathrm{o2} < o$ then
            $(o, v) \leftarrow (\mathrm{o2}, \mathrm{v2})$                ▷ New solution accepted
        else
            $i \leftarrow i + 1$
            Undo application of neighbourhood $n$
        end if
    end while
end procedure
procedure REPAIR
    $o \leftarrow$ current objective
    $v \leftarrow$ total violation
    $i \leftarrow 0$                                  ▷ iterations spent without improving
    $\sigma \leftarrow$ available neighbourhoods
    while $i \leq$ limit do                           ▷ limit is a tunable parameter
        select a random neighbourhood $n \in \sigma$
        $(\mathrm{o2}, \mathrm{v2}) \leftarrow$ RUNNEIGHBOURHOOD($n$)
        if $\mathrm{v2} \leq v$ then
            $(o, v) \leftarrow (\mathrm{o2}, \mathrm{v2})$                ▷ New solution accepted
        else
            $i \leftarrow i + 1$
            Undo application of neighbourhood $n$
        end if
    end while
end procedure
procedure CLIMB
    $o \leftarrow$ current objective
    $v \leftarrow$ total violation
    $i \leftarrow 0$                                  ▷ iterations spent without improving
    $\sigma \leftarrow$ available neighbourhoods
    while $i \leq$ limit do                           ▷ limit is a tunable parameter
        select a random neighbourhood $n \in \sigma$
        $(\mathrm{o2}, \mathrm{v2}) \leftarrow$ RUNNEIGHBOURHOOD($n$)
        if $v = 0 \wedge \mathrm{o2} \leq o$ then
            $(o, v) \leftarrow (\mathrm{o2}, \mathrm{v2})$                ▷ New solution accepted
        else
            $i \leftarrow i + 1$
            Undo application of neighbourhood $n$
        end if
    end while
end procedure
```

The solver runs in a loop: for each iteration, a random neighbourhood is executed and the change to the total constraint violation and objective are examined. If the violation count is decreased or if the objective is improved, the new solution is accepted. Otherwise, the change is reversed. This means that until the set of assignments form a feasible solution, the solver will accept any change to the objective (better or worse) provided that the set of assignments is brought closer to a feasible solution. As mentioned previously, the distance from a feasible solution is a heuristic. It is the total constraint violation count.

Once a feasible solution is found (the violation count reaches 0), the solver may only make progress by making changes that do not worsen the objective. If after a number of iterations [1] no improvement is observed, the solver relaxes the restriction on the violation count; accepting solutions if they improve the objective and allowing some constraints to be violated. In general, this procedure can be considered as being similar to hill climbing search procedures commonly found in local search solvers. Future work would focus on extending the set of search strategies to include other popular methods such as Simulated Annealing [16] or Tabu Search [14].

7 Case Studies

Wwe present experiments on three problem classes. Referring to our hypothesis in Section 1.1, our experiments show the performance of Athanor against other local search solvers that also automatically derive neighbourhoods:

- OscaR/CBLS [6]. Like Athanor, this approach derives neighbourhoods automatically through analysis of the input problem. However, the input in this case is a constraint model rather than an abstract Essence specification.
- Two variants of large neighbourhood search (LNS), propagation guided [21] and explanation based [23] (referred to as PG-LNS and EB-LNS respectively). These dynamically derive neighbourhoods by tracing either propagations or failures during the search for a solution to a constraint model.
- SNS [3]. This approach also derives neighbourhoods from an Essence specification, but refines the specification and the derived neighbourhoods to a lower level representation before search.

Input to OscaR/CBLS was produced by using Conjure to refine Essence specifications, Savile Row to transform the models into MiniZinc [17], and MiniZinc 2.1.7 to specialise for OscaR/CBLS's Flatzinc backend. The two variants of LNS were implemented using the Choco 4.0.6 solver using models that were produced by hand. Care was taken to match the SNS, Choco and OscaR/CBLS models as closely as possible. Details are included with each case study.

[1] this is a tunable parameter to the solver

Table 1: Minimising objective, after 10 seconds and 600 seconds. Best results for each instance and time period are given in bold.

Instance	Athanor		SNS		EB-LNS		PG-LNS		OscaR/CBLS	
	10s	600s	10s	600s	10s	600s	10s	600s	10s	600s
sonet1	**66.5**	62.5	170.0	**60.5**	72.5	65.0	85.0	72.5	75.5	72.5
sonet2	**182.5**	**117.0**	578.5	126.0	216.5	143.5	281.0	123.5	546.0	157.5
sonet3	**115.5**	**91.5**	348.0	95.5	132.5	104.5	153.0	101.0	256.5	121.5
sonet4	**184.0**	132.0	578.0	**124.5**	234.0	148.5	283.5	133.5	531.5	167.0
sonet5	**258.5**	**166.5**	833.5	171.0	394.5	199.0	474.5	176.0	843.0	227.5
sonet6	**294.0**	**178.0**	796.5	191.0	391.5	227.0	490.0	188.0	829.5	259.5
sonet7	**370.5**	**212.0**	1102.5	282.5	534.0	285.5	956.0	253.5	1152.5	319.0
sonet8	**355.5**	**197.0**	1334.0	274.0	704.5	261.5	1236.5	236.0	1379.0	295.0

For each problem class, we present a table of results for a selection of instances, some randomly generated and some from benchmarking libraries. For each solver, the objective achieved after ten seconds and after ten minutes is shown, and these values are the median of 10 runs.

7.1 Case Study 1: SONET

As mentioned previously, the strength of operating directly on Essence types is that the Essence type constructors can convey information on the structure of the problem being solved. Referring to the Sonet problem presented in Figure 1, notice there is only one abstract variable.

```
find network :
  mset (size nrings) of set (maxSize capacity) of Nodes
```

However, this variable has a very descriptive type which would be hard to reconstruct from the myriad representations that may be used to encode this abstract variable for input to a low level CP or local search solver. The `size` attribute on the outermost type (multiset) forces the multiset to have a fixed size. Athanor creates the multiset with the correct size, and maintains the size as an invariant, which allows Athanor to drop all neighbourhoods that would change the size of the multiset. Athanor only generates neighbourhoods that manipulate the elements of the multiset. Since the inner type is a set of variable size, the neighbourhoods `setAdd`, `setRemove` are used. Athanor also uses the neighbourhoods `setSwap`, which exchanges one element in a set for another and `setAssignRandom`, which assigns an entire set to a new value. The `setSwap` and `setRemove` neighbourhoods are biased towards selecting the sets which are most violating, and then the most violating elements of those sets. When refining the model for input to the low level solvers, a standard representation was chosen. Each set in the multiset was encoded using the occurrence representation; a matrix of Booleans determining whether or not each item was present in the set. The multiset was encoded explicitly, one matrix of booleans for each set in the multiset. Symmetry breaking constraints were not included as Prestwich [22] suggests that such constraints can harm the performance of local search.

Table 1 shows the median optimisation value found after ten seconds and ten minutes. As can be seen in Table 1 our solver always achieves the best performance after ten seconds, and is best or second after Structured Neighbourhood Search (SNS) [3] when ten minutes have elapsed.

Figure 5 shows the median objective of each solver against time. This demonstrates how Athanor rapidly climbs towards solutions despite the simple hill-climbing search procedure. In the figure, black represents Athanor, blue represents explanation guided LNS, red represents propagation guided LNS, green represents SNS and orange represents OscaR/CBLS.

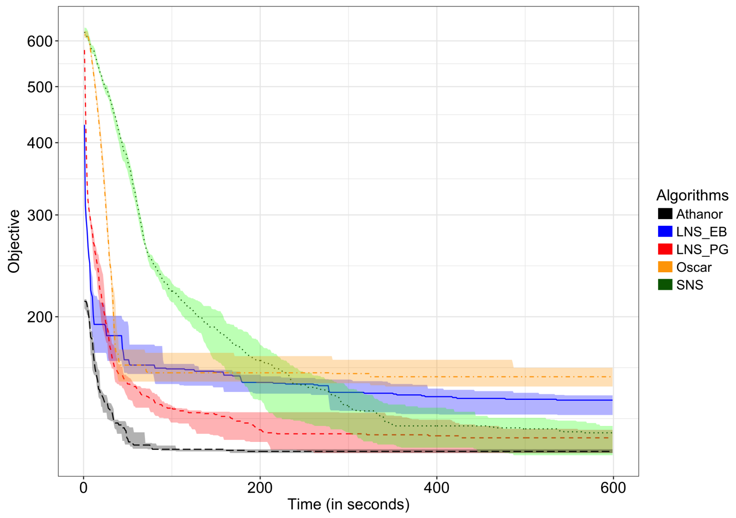

Fig. 5: Plot of progress solving instance sonet2, showing progress on minimizing the objective against time. Lines indicate the median objective value and the shaded region around each line indicates the interquartile range.

7.2 Case Study 2: The Knapsack Problem

Figure 6 shows an Essence specification of the Knapsack problem. The variable:

```
knapsack : set of object
```

has less structure than the `network` variable from Sonet, discussed in Section 7.1. The simplicity of the variable structure and constraints in the Knapsack

```
given object new type enum
given weight, value : function (total) object --> int(1..)
letting maxWeight be max([w | (_,w) in weight])

find knapsack : set of object
maximising sum i in knapsack . value(i)
such that sum i in knapsack . weight(i) <= maxWeight
```

Fig. 6: An Essence specification of the Knapsack Problem

problem mean that other solvers that only accept primitive (integer or Boolean) variables are at less of a disadvantage. Despite this, Athanor still performs extremely well, finding the best result in most experiments. Once again, the standard low level occurrence representation was chosen for input to the other solvers. Table 2 shows the performance of the different solvers on knapsack instances of size 1000 and 5000, given ten seconds and ten minutes. Athanor scales well, always winning on the largest instances.

Although LNS performed well with the Sonet problem, it performs significantly worse than OscaR/CBLS and our solver with the Knapsack problem on larger instances. However, though OscaR/CBLS performed less well on the Sonet problem, it is clearly able to leverage the simplicity of the Knapsack problem. Despite this, Athanor is competitive and is able to find better objectives in seven out of the ten instances tested.

7.3 Case Study 3: The Golomb Ruler Problem

```
language Essence 1.3

given n : int(1..)
letting bound be 2 ** n

find Ticks : set (size n) of int(0..bound)
minimising max(Ticks)

such that
0 in Ticks,
forAll {i, j} subsetEq Ticks .
        forAll {k, l} subsetEq Ticks .
            {i, j} != {k, l} -> i - j != k - l
```

Fig. 7: An Essence specification of the Golomb Ruler Problem

Table 3 shows that ATHANOR does not perform well with this model of the Golomb Ruler. There are two main reasons for this poor performance. Firstly, the only variable in the model is a fixed size set. The lack of a nested structure means that ATHANOR's advantage over other solvers is limited. In the current version of ATHANOR only one neighbourhood is used, which selects the most violating integer and randomly changes it. An explicit representation was chosen for the low level refinements given to the other solvers due to the set having a large domain, exponential relative to the size of the Golomb ruler.

Secondly, as explained in Section 5, the AST template representing the expression under a quantifier is copied for every element that is unrolled. The expression `forAll {i, j} subsetEq Ticks` unrolls to approximately n^2 expressions. The further nested quantifier `forAll {k, l} subsetEq Ticks` also unrolls to approximately n^2 expressions. This results in approximately n^4 copies of the innermost expression `{i, j} != {k, l} -> i-j != k-l` and hence, around n^3 parents triggering whenever any element of the set is changed. This results in the relatively poor performance observed.

This can be greatly improved. Note that once i, j, k, l are instantiated with values across the n^4 expressions, there are around n^2 duplicates of each copy of $j - i$ and $k - l$. Rather than having many AST nodes representing $i - j$ for the same values of i and j, a single AST node ought to be reused. This optimisation is used in the models used in the other solvers. The difficulty compared to the other solvers is that the duplication must be detected while loops are dynamically unrolled. Even if all duplicate expressions were eliminated, there would still be n^2 parents that would need to be triggered on every change to the set. This is due to the clique of disequalities that is enforced on all the tick differences `i - j != k - l`. Further replacing these disequalities with a single `allDifferent` constraint would reduce the number of parents (and consequently the number of trigger events per element change) to n. Both of these improvements are targets for future work.

8 Conclusion

We have presented the benefits of having a solver operate directly on the abstract variables present in an ESSENCE specification. The solver is able to utilise the type system to construct effective neighbourhoods and is able to model the abstract types directly without having to resort to refinements into primitive types that may be very large. We have presented a framework for incremental evaluation of ESSENCE ASTs as the values taken by the abstract variables change during search. This includes the dynamic unrolling of quantifiers as the size of containers (such as sets) changes during search. We benchmarked the proof of concept solver ATHANOR using three problem classes containing the ESSENCE types multiset and set.

Acknowledgements We thank EPSRC for grants EP/P015638/1 and EP/P026842/1. Jefferson holds a Royal Society University Research Fellowship.

Table 2: Knapsack Problem: Maximising objective, after 10 seconds and 600 seconds. Best results for each instance and time period are given in bold.

Instance	Athanor		SNS		EB-LNS		PG-LNS		OscaR/CBLS	
	10s	600s	10s	600s	10s	600s	10s	600s	10s	600s
1000-1	**550791.0**	**550791.0**	337699.0	496282.0	361345.5	455680.5	349476.5	400570.0	341029.5	341029.5
1000-2	**207520.5**	**209583.0**	168921.5	206756.0	173783.5	192808.0	171055.5	181423.0	165726.0	165726.0
1000-3	**344296.0**	349968.0	315976.0	347283.0	296269.0	369869.0	286926.0	341746.0	327795.0	**377124.0**
1000-4	137901.0	140015.0	110904.5	148802.0	103526.0	154480.5	97978.0	136409.5	**144588.5**	**162715.0**
1000-5	315920.0	318920.0	**225452.0**	**337268.0**	214114.0	333684.0	200038.0	314428.0	317772.0	**337268.0**
5000-1	**922018.5**	**922018.5**	337699.0	496282.0	921765.0	921870.0	921724.5	921990.0	921294.0	921294.0
5000-2	**1164516.0**	**1164576.0**	168921.5	206756.0	1164496.5	1164663.0	1164466.5	1164720.0	1164235.5	1164235.5
5000-3	**1414441.5**	**1414542.0**	315976.0	347283.0	1251067.5	1264284.0	1247836.5	1253829.0	1277577.0	1414386.0
5000-4	**1667823.0**	**1667895.0**	110904.5	148802.0	1254748.5	1268286.0	1257783.0	1263609.0	1286335.5	1667565.0
5000-5	**1889281.5**	**1889355.0**	225452.0	337268.0	1237038.0	1250977.5	1237560.0	1243005.0	1288674.0	1888881.0

Table 3: Golomb Ruler Problem: Minimising objective, after 10 seconds and 600 seconds. Best results for each instance and time period are given in bold.

Instance	Athanor		SNS		EB-LNS		PG-LNS		OscaR/CBLS	
	10s	600s	10s	600s	10s	600s	10s	600s	10s	600s
n10	109.0	66.5	830.0	577.5	**64.0**	64.0	81.5	77.0	70.0	**60.0**
n11	217.0	90.0	1596.0	1241.0	**85.5**	79.5	105.0	105.0	95.0	**78.5**
n12	810.0	132.5	3278.0	2470.5	275.5	103.0	122.0	120.5	**119.0**	**100.5**
n13	5087.0	191.5	6807.5	5446.5	4332.0	**133.5**	1387.5	151.0	**164.5**	134.5
n14	13823.0	306.5	16384.0	16384.0	11209.0	**160.0**	10436.0	185.5	**192.5**	174.0
n15	28641.5	634.5	32768.0	32768.0	23569.5	**208.0**	28053.0	246.0	**220.0**	220.0
n16	61660.0	3701.0	65536.0	65536.0	62371.0	52232.5	62371.0	564.5	**250.5**	**250.5**
n17	Inf	Inf	131072.0	131072.0	**125217.5**	61851.5	**125217.5**	**36325.0**	Inf	Inf

References

1. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
2. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–116. Springer (2013)
3. Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A., Spracklen, P.: A framework for constraint based local search using Essence. In: IJCAI. pp. 1242–1248 (2018)
4. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI. pp. 3–8 (2014)
5. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011)
6. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**(3), 325–345 (2015). https://doi.org/10.1007/s10601-015-9184-z
7. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: Proc. of the CP-01 Workshop on Modelling and Problem Formulation. p. 223 (2001)
8. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling: Exploiting common patterns in constraint programming. In: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems. pp. 27–41 (2002)
9. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The Essence of Essence. Modelling and Reformulating Constraint Satisfaction Problems pp. 73–88 (2005)
10. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of Essence: A constraint language for specifying combinatorial problems. In: IJCAI. pp. 80–87 (2007)
11. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
12. Frisch, A.M., Hnich, B., Miguel, I., Smith, B.M., Walsh, T.: Transforming and refining abstract constraint specifications. In: International Symposium on Abstraction, Reformulation, and Approximation. pp. 76–91. Springer (2005)
13. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. vol. 141, pp. 98–102 (2006)
14. Glover, F., Laguna, M.: Tabu search. In: Handbook of combinatorial optimization, pp. 2093–2229. Springer (1998)
15. Hentenryck, P.V., Michel, L.: Constraint-based local search. The MIT press (2009)
16. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
17. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP. pp. 529–543. LNCS 4741, Springer (2007)

18. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: CP. pp. 590–605. LNCS 8656, Springer (2014)
19. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
20. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: CP. pp. 330–340. LNCS 9255, Springer (2015)
21. Perron, L., Shaw, P., Furnon, V.: Propagation guided large neighborhood search. In: CP. pp. 468–481. LNCS 3258, Springer (2004). https://doi.org/10.1007/978-3-540-30201-8_35, `https://doi.org/10.1007/978-3-540-30201-8_35`
22. Prestwich, S.: Negative effects of modeling techniques on search performance. Annals of Operations Research **118**(1), 137–150 (Feb 2003). https://doi.org/10.1023/A:1021809724362, `https://doi.org/10.1023/A:1021809724362`
23. Prud'homme, C., Lorca, X., Jussien, N.: Explanation-based large neighborhood search. Constraints **19**(4), 339–379 (2014). https://doi.org/10.1007/s10601-014-9166-6, `https://doi.org/10.1007/s10601-014-9166-6`
24. Russell, S., Norvig, P.: Artificial Intelligence A Modern Approach, Third Edition. Pearson (2014)
25. Smith, B.M.: Symmetry and search in a network design problem. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 336–350. Springer (2005)