A Framework for Constraint Based Local Search using ESSENCE

Özgür Akgün, Saad Attieh, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, András Z. Salamon, Patrick Spracklen, James Wetter
School of Computer Science, University of St Andrews, UK
{ozgur.akgun, sa74, ian.gent, caj21, ijm, pwn1, Andras.Salamon, jlps}@st-andrews.ac.uk

Abstract

Structured Neighbourhood Search (SNS) is a framework for constraint-based local search for problems expressed in the ESSENCE abstract constraint specification language. The local search explores a structured neighbourhood, where each state in the neighbourhood preserves a high level structural feature of the problem. SNS derives highly structured problem-specific neighbourhoods automatically and directly from the features of the ESSENCE specification of the problem. Hence, neighbourhoods can represent important structural features of the problem, such as partitions of sets, even if that structure is obscured in the low-level input format required by a constraint solver. SNS expresses each neighbourhood as a constrained optimisation problem, which is solved with a constraint solver. We have implemented SNS, together with automatic generation of neighbourhoods for high level structures, and report high quality results for several optimisation problems.

1 Introduction

Constraint Programming (CP) offers an efficient means of solving complex constrained optimisation problems. A systematic search through the space of partial assignments may be employed to achieve this. In local search methods [Hoos and Stützle, 2004], a complete assignment is iteratively modified in an effort to reach a satisfying or optimal solution. At each iteration a modification is selected from a *neighbourhood* of available candidates. Local search sacrifices completeness for the ability to explore neighbourhoods rapidly.

Constraint-based local search (CBLS) [Codognet and Diaz, 2001; Van Hentenryck and Michel, 2005] has applied local search to CP. Neighbourhoods are specified by the user as part of the model, or generated from the constraints and variables in the model [Benoist *et al.*, 2011; Björdal *et al.*, 2015]. A potential weakness of this approach is that a constraint model typically represents the abstract structure of a given problem as a collection of primitive variables and constraints. Focusing on variables and constraints individually may produce weaker neighbourhoods than if the original problem structure were apparent. This observation motivates our work: by allowing neighbourhoods to exploit the original problem's combinatorial structure, we expect local search to attain better performance than when problem structure is not available.

The technique of Large Neighbourhood Search (LNS) originated in the constraint community [Shaw, 1998], where it has been further refined to exploit propagation [Perron *et al.*, 2004] and explanations [Prud'homme *et al.*, 2014]. There are also customisable and hybrid LNS approaches [Rendl *et al.*, 2015; Cipriano *et al.*, 2013]. Pesant and Gendreou extended LNS to support meaningful neighbourhoods (e.g. exchanging two cities in the Travelling Salesperson Problem) [1999]. The advance we present here is that SNS is able to automatically generate neighbourhoods from a high level specification which are semantically meaningful in the original problem.

For high level problem expression we use the ESSENCE constraint specification language [Frisch *et al.*, 2005; 2007; 2008]. Space precludes a detailed presentation, but Fig. 1 shows an illustrative specification of the capacitated vehicle routing problem (CVRP). 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`). Here, the combinatorial object to be found is represented by a single abstract variable whose type is `set of sequence of int`. This is a considerably higher level of description than previous work on CBLS described as working from high-level models [Van Hentenryck and Michel, 2007]. Ågren et al. extend local search to support sets of integers directly [2009]. We improve over this by supporting arbitrarily nested types. Furthermore, we avoid having to add explicit support for these types to the underlying constraint solver.

ESSENCE is situated at the top of a constraint modelling pipeline, which automatically models and solves an ESSENCE specification. The CONJURE automated modelling system [Akgün *et al.*, 2011; 2013] employs refinement rules to convert an ESSENCE specification into a lower-level but generic language ESSENCE PRIME [Rendl, 2010]. This model is automatically prepared for a given constraint solver while performing solver-specific model optimisations by SAVILE ROW [Nightingale *et al.*, 2014; 2015; 2017].

By using ESSENCE as the starting point we can exploit the structure in ESSENCE specifications, avoiding the diffi-

```
language Essence 1.3
$ Simple version of capacitated vehicle routing problem
$ Goods are shipped from one depot to drop off locations
$ One order per drop off location, one trip per vehicle
given N : int                       $ Number of locations
letting L0 be domain int(0..N)      $ Depot is 0
letting L1 be domain int(1..N)
given weights : function (total) L1 --> int(1..)
given costs : function (total) tuple (L0,L0) --> int(0..)
given vehicleCapacity : int         $ All have same limit

letting totalW be sum([weight | (_,weight) <- weights])
letting minVehicles be totalW / vehicleCapacity
  + toInt(totalW % vehicleCapacity != 0)

find plan : set (minSize minVehicles, maxSize N)
            of sequence (maxSize N, injective, minSize 1)
            of L1

$ Worst-case: separate vehicle out and back for each item
find optVar : int( 0 .. sum([costs((0, i)) | i : L1])*2 )
such that optVar = sum route in plan . (
  sum([ costs(tuple(route(i-1), route(i)))
      | i : int(2..N), i <= |route| ])
  + costs((0, route(1)))  $ from depot to first location
  + costs((route(|route|), 0)) ) $ ...and back from last
minimising optVar

$ Capacity restriction, every order delivered
such that forAll route in plan . vehicleCapacity >=
  (sum (_,order) in route . weights(order))
such that forAll order : L1 . 1 =
  sum([toInt(order = loc) | route<-plan, (_,loc)<-route])
```

Figure 1: Capacitated vehicle routing in ESSENCE

cult task of identifying that structure in an equivalent constraint model. Neighbourhoods are automatically generated as constraint problems describing transformations of members of ESSENCE domains via a library of domain specific operations that encapsulate expert knowledge. The resulting method, which we call Structured Neighbourhood Search, retains the advantages of specifying problems in ESSENCE — concision, abstraction from modelling details and blackbox search — while actually *improving* the quality of the neighbourhoods we can generate for constraint-based local search.

2 An Overview of SNS

To help the reader follow our exposition we start with a high-level overview of Structured Neighbourhood Search. The key point is that the use of ESSENCE allows neighbourhoods explored by the low-level search to be *semantically meaningful* in the high-level specification of the problem. For example, in a problem concerning sets, SNS can ensure that consecutive states in the local search represent valid sets, even using a low-level solver which is not aware of the structure of sets. To achieve this, CONJURE analyses a specification and automatically generates semantically meaningful neighbourhoods (Sec. 3). The pipeline then refines these neighbourhoods so that they can be used by the SNS solver. The SNS solver (Sec. 4) then performs a relatively standard local search, selecting appropriate neighbourhoods automatically using a multi-armed bandit algorithm (Sec. 4.4). Within each neighbourhood, a complete constraint solver is used to ensure that the structural constraints defining the neighbourhood are obeyed. As a result, SNS combines the large neighbourhoods of LNS, the high-level semantic structure (without the need for a specially-written search procedure), and the efficiency of modern constraint solvers at the low-level.

3 Automatic Neighbourhood Generation

To date, our pipeline has targeted systematic constraint and SAT solvers as output [Nightingale *et al.*, 2017]. In this paper we describe modifications to the pipeline to support SNS. Our main contribution is that for each abstract decision variable in the original specification, a set of neighbourhoods is generated automatically according to a set of neighbourhood generation rules. The strength of these neighbourhoods is that they encode powerful operations derived from the structure of the abstract variable, which provide strong guidance for the SNS search process. These neighbourhoods would be both difficult and costly to generate from the equivalent collection of primitive variables and constraints if a constraint model were the starting point because the abstract structure (e.g. a sequence, set or function) would have to be recognised.

The CVRP specification in Fig. 1 uses a single abstract variable `plan`, modelled as a `set of sequence of int`. Neighbourhoods we generate for this variable include: add or remove sequences to/from the set; select a sequence from the set and reverse a contiguous subsequence; select two sequences s_1 and s_2 and for a range of indices, swap the values contained by s_1 at those indices with the values contained by s_2 at the same indices. This example illustrates the advantage of SNS: the `set of sequence of int` type is explicit in the ESSENCE specification, and can be exploited to automatically generate high quality neighbourhoods.

A *neighbourhood specification* extends the original ESSENCE specification of a problem with additional abstract variables and constraints to express each of the neighbourhoods to be explored for this problem. Each such neighbourhood is generated automatically by CONJURE. We illustrate this process by means of a concrete example, focusing on the variable type `set of sequence of int` and the neighbourhood mentioned above: select a sequence from the set and reverse a contiguous subsequence.

We refer to the abstract variables in the original ESSENCE specification as *primary variables*. Each neighbourhood is expressed as a set of constraints relating these primary variables to an *active solution*, which is a feasible solution to the problem. The first step in neighbourhood generation is for each primary variable to introduce a corresponding *active variable* to store the currently active solution. For the example given in Fig. 1, CONJURE introduces a new active variable `active_plan` corresponding to the primary variable `plan`:

```
find active_plan : set (minSize minVehicles, maxSize N)
            of sequence (maxSize N, injective, minSize 1)
            of L1
```

The next step is to add variables local to each neighbourhood. All neighbourhoods have an associated *size* variable, which control the size of the neighbourhood explored. Further variables may be introduced to control other such parameters. In our example, an additional variable is introduced to indicate the starting index of the subsequence to be reversed:

```
find revSubSize : int(1..N)
find subStart   : int(1..N)
```

The final step is to express each neighbourhood by adding constraints between primary and active variables. In our example below, CONJURE adds constraints to the specification constraining the value of the primary variable `plan` to be the same as that held by `active_plan` except that one of the sequences in `plan` must have a contiguous portion reversed.

```
$ select a primary sequence
exists s_p in plan .
  $ select a sequence in the active solution
  exists s_a in active_plan .
    $ Enforce subsequence indices are in range
    subStart + revSubSize <= |s_a|,
    $ reverse the selected subsequence
    forAll i : int(0..N) .
      i <= revSubSize ->
      s_p(subStart + i) = s_a((subStart + revSubSize) - i)
    $ Regions outside subsequence are equal to active sol
    |s_a| = |s_p|,
    forAll i : int(1..N) .
      (i <= |s_a| /\
      (i < subStart \/ i > subStart + revSubSize)) ->
      s_p(i) = s_a(i)
    $ Ensure all other sequences are unchanged
    plan = active_plan - s_a + s_p,
```

There are typically a variety of neighbourhoods in each neighbourhood specification. In order that the SNS search process can choose which to employ at any particular time, each is enabled with a Boolean *activator* variable.

3.1 Neighbourhood Generation Rules

Our neighbourhood generation rules are driven by the high-level nested structure of ESSENCE types. *Direct* rules (Table 1) are concerned solely with the outer structure of an ESSENCE decision variable. For instance, we might wish to reverse a contiguous subsequence of a `sequence of` τ, where τ is any ESSENCE type. *Lifted* rules are parameterised by other neighbourhood generation rules. They allow us to reach inside a containing abstract structure and apply other rules to manipulate individual parts of that structure. This was illustrated in our example, where we selected and reversed a subsequence of a single sequence from a set of sequences.

Lifted rules can themselves be lifted, allowing us to generate a large variety of neighbourhoods by composing a small number of neighbourhood generation rules in multiple ways.

Powerful neighbourhoods can be constructed to make multiple simultaneous changes, which must be coordinated so that the modifications made are valid. For example, given a set of sequences we may wish to exchange elements between sequences, hence modifying the value of two sequences within the set simultaneously. These *Synchronised* rules (Table 2) can both operate directly or be lifted.

4 Structured Neighbourhood Search

The process described in the previous section generates a neighbourhood specification, which is automatically refined by CONJURE into a constraint model in ESSENCE PRIME. ESSENCE PRIME is a solver-independent constraint modelling language offering features similar to those of MiniZinc [Nethercote *et al.*, 2007] or OPL [Van Hentenryck *et al.*, 1999], with support for Boolean and integer decision variables together with arithmetic, logical and global constraints. This model is annotated by CONJURE to indicate the parts corresponding to each neighbourhood then processed by SAVILE ROW into input suitable for the SNS solver.

`set` (and `multiset`)
Remove s elements from set.
Add s elements to set.
Exchange s elements for values not currently in set.
ExchangeOrRemove a total of s elements from set.
ExchangeOrAdd a total of s elements to set.
`sequence`
Reverse contiguous subsequence of length s.
Exchange the positions of s elements with another s.
Relax domains of contiguous subsequence, length s.
CycleL/R elements of a sequence s positions left/right.
RemoveE/S s elements from the end/start of a sequence.
AddE/S s elements to the end/start of a sequence.
`function`
Define values for s more elements of the function domain.
UnDefine values for s elements of the function domain.
Injective- Select $2s$ elements from the image of a function: reduce to s elements by constraining pairs of elements in the preimage to be equal.
Injective+ Select s elements from the image of a function: increase to $2s$ elements by constraining pairs of elements in the preimage to be distinct.
Permute the images of s inputs to the function.

Table 1: *Direct* neighbourhood generations rules for size s. Each of these can be *Lifted* to operate on nested types.

`set` (and `multiset`)
Move s elements from one set to another.
Exchange s elements between two sets.
Merge a set with another set of size s.
Split s elements from a set to create a new set.
`sequence`
Concatenate one sequence with another.
Exchange s elements between two sequences.
Split s elements from a sequence to create a new sequence.
CrossOver Exchange s contiguous elements between two sequences at the same indices.
Move Remove s elements from the end of a sequence; add those elements to the start or end of another sequence.
`function`
CrossOver Exchange s elements from the image of one function with s elements from the image of another.

Table 2: *Synchronised* neighbourhood generation rules for size s. These rules can be used directly on a single variable or can be lifted to operate on parts of an abstract structure.

Algorithm 1 SNS()

```
$ ι: the incumbent (best solution found so far)
ι ← SNS-Improve(SNS-FindRandSoln())
$ Each neighbourhood has an associated size
s ← 1
while true do
    for n ∈ neighbourhoods in random order do
        Set size of n to s
        σ ← SNS-Neighbourhood-Search(ι, n, Explore)
        σ ← SNS-Improve(σ)
        if σ improves upon ι then
            s ← 1, ι ← σ
            Continue while loop
    if 2 * s ≤ maximum neighbourhood size then
        s ← 2 * s
    else
        repeat
            σ ← SNS-Improve(SNS-FindRandSoln())
        until σ improves upon ι
        s ← 1, ι ← σ
return ι on timeout
```

4.1 The SNS Search Procedure

Algorithm 1 summarises the SNS search procedure. The first step is to find a random feasible solution (§4.2). The random feasible solution is improved by applying *SNS-Improve()* and it becomes the first incumbent solution ι. The variable s represents the neighbourhood size used for diversification, and $s = 1$ initially. The first part of the main loop picks a neighbourhood and applies it to ι (with size s) to move away from ι, creating a new active solution σ that may be worse than ι. *SNS-Improve()* is applied to improve σ. If the result is better than ι it is accepted as the new incumbent, and s is reset to 1.

In the case where the set of neighbourhoods is exhausted, s is increased and each neighbourhood is applied again. The diversification process is repeated until s cannot be increased further. Finally, the algorithm will generate new random feasible solutions (unrelated to ι), apply *SNS-Improve()* to each one, and accept the first that improves on ι. In the following subsections we describe the constituent parts of the algorithm.

4.2 Finding Random Feasible Solutions

At two points *SNS()* requires the production of random feasible solutions (function *SNS-FindRandSoln()*): to find the initial incumbent and to escape local optima when the maximum neighbourhood size has been reached. For this purpose, we employ the MINION constraint solver [Gent *et al.*, 2006] with the *dom/wdeg* variable ordering [Boussemart *et al.*, 2004], a random value ordering and random restarts, beginning at 100ms and increasing by a multiple of 1.5 each iteration.

4.3 Finding Improved Solutions

Algorithm 2 *SNS-Improve()* is called with a feasible solution σ, the active solution. We select a neighbourhood (§4.4) and run the neighbourhood search procedure (§4.5) in optimisation mode. If a solution σ' that is at least as good as σ is found, σ' is made the new active solution and the process is repeated. If no σ' of sufficient quality is found within a given

Algorithm 2 SNS-Improve(Active solution σ)

```
peakThreshold ← α
increment ← β× 1 / number of neighbourhoods
while RandomValueIn({0 . . . 1}) ≥ peakThreshold do
    n ← SNS-Select-Neighbourhood()
    σ' ← SNS-Neighbourhood-Search(σ, n, Optimise)
    if σ' is not worse than σ then
        peakThreshold ← α
        σ ← σ'
    else
        peakThreshold ← peakThreshold + increment
return σ
```

timeout, we add an *increment* to the *peakThreshold*, which is used to decide probabilistically if we have reached a local optimum. In this paper we set α to 0.001 and β to $\frac{1}{16}$. Future work will explore tuning these parameters.

4.4 Neighbourhood Selection

The *SNS-Select-Neighbourhood()* procedure (called from Algorithm 2) is used to find the most promising neighbourhood at any point during search. For a given problem there may be many generated neighbourhoods for which we have no prior knowledge of their reward distributions, determined by how much the application of each neighbourhood improves the objective. As we have no prior knowledge of the suitabilities of particular neighbourhoods to a particular problem class, during search we have to carefully balance the time taken exploring and identifying a neighbourhood's performance while exploiting those that allow us to rapidly improve. Representing this as a multi-armed bandit problem allows us to employ well known regret minimising algorithms to deal with the exploration/exploitation dilemma. This can be seen as a form of 'Adaptive LNS' [Ropke and Pisinger, 2006].

The multi-armed bandit can be seen as a set of real distributions, each distribution being associated with the rewards delivered by one of the *K* levers. In our case this is the *K* generated neighbourhoods. On each iteration of *SNS-Improve()* one neighbourhood is selected to locally search the problem space and a reward is observed based upon the improvement to the active solution. Our aim is at each iteration to apply the optimal neighbourhood, where optimality is defined as producing the largest increase in the value of the objective. The regret ρ after *T* rounds is defined as the expected difference between the reward sum associated with an optimal strategy and the sum of the collected rewards observed. The UCB1 [Auer *et al.*, 2002] algorithm was chosen to solve the multi-armed bandit problem as first and foremost its regret grows logarithmically in line with the number of actions taken. For each neighbourhood k we record the average reward x^k and the number of times k has been tried in the search procedure n_j out of a total of n iterations. On each iteration a neighbourhood is chosen that maximises $x_j + \sqrt{2\log(n)/n_j}$.

In our system the reward distributions for a neighbourhood are not fixed, so this is not a Stationary Multi-Armed Bandit problem. However, if a neighbourhood performs well, we expect it will continue performing well during search even if there is a slight variation in the mean reward. We have

found that using UCB1 gives good results. Future work could investigate the use of Upper Confidence Bound policies for non-stationary bandit problems, such as the family of Exp3 algorithms [Kocsis and Szepesvári, 2006; Munos, 2014].

4.5 Neighbourhood Search

SNS-Neighbourhood-Search(σ, n, Mode) is responsible for searching the neighbourhood n of the active solution σ. It can be run in two *Mode*s. The first (*Explore*) accepts the first solution of any objective value. This is used by *SNS()* in Algorithm 1. The second mode (*Optimise*, employed by *SNS-Improve()*) treats the neighbourhood as a constrained optimisation problem, accepting only those solutions with an objective value at least as good as that of the active solution σ. Once *Optimise* finds a solution, it will continue to search for a solution with better objective value, until the time limit of 500ms is reached. In preliminary experiments we found a time limit more effective than a search node limit as the latter penalises simple neighbourhoods that have high node rates.

Neighbourhood search is performed using the systematic MINION solver in a carefully controlled manner. The annotations supplied by CONJURE are used to identify the primary and active variables involved in neighbourhood n. The primary variables are unassigned preparatory to search, and the activator variable for neighbourhood n is set to true, removing the guard on the neighbourhood constraints.

We construct a variable ordering as follows. First is the neighbourhood size variable for n, which is given an ascending value ordering. This forces the exploration to begin close to the current active solution σ before systematically expanding out. Next are placed the variables that control the neighbourhood parameters, e.g. subStart in our example. A random value ordering is used so that different portions of the neighbourhood are searched on each activation. Finally come the primary variables upon which the neighbourhood is operating, that is the representation of a partition, set, sequence, etc. These variables are searched using a *dom/wdeg* ordering with the aim of finding solutions as quickly as possible once the neighbourhood parameters have been chosen.

5 Experimental Evaluation

Our hypothesis is that the structure in abstract ESSENCE specifications can be exploited to automatically generate effective neighbourhoods for constraint-based local search. The most natural comparison is with methods that generate neighbourhoods from the constraint model. Therefore we compare SNS with OscaR/CBLS [Björdal *et al.*, 2015], a modern implementation of CBLS, propagation-guided LNS (LNS PG) [Perron *et al.*, 2004], and explanation-based LNS (LNS EB) [Prud'homme *et al.*, 2014] (which are automatic large neighbourhood search algorithms). LocalSolver [Benoist *et al.*, 2011] is another possible point of comparison, but this commercial solver incorporates a complex mix of technologies that make it unsuitable to test our hypothesis. We use Chuffed (via its free search option), a systematic lazy clause generation solver [Chu, 2016], as a performance baseline.

We compare SNS to the other solvers on 5 problem classes: Capacitated Vehicle-Routing Problem (CVRP); Progressive Party Problem (PPP, CSPLib 13); Minimum Energy Broadcast (MEB, CSPLib 48); SONET (CSPLib 56); Rack Configuration Problem (RackConf, CSPLib 31).

For SNS, each problem class and a set of instances is specified in ESSENCE. An extended version of CONJURE is applied once per problem class to generate a model in ESSENCE PRIME with SNS neighbourhoods. An extended version of SAVILE ROW [Nightingale *et al.*, 2017] is applied for each instance to instantiate the model and specialise it in two ways: SNS (including neighbourhoods) and MiniZinc (with neighbourhoods removed). The MiniZinc instance is then specialised via MiniZinc 2.1.7 for OscaR/CBLS and Chuffed.

For the LNS methods, each problem class was implemented in Choco 4.0.6, with great care taken to match the ESSENCE PRIME model as closely as possible. For the search within each LNS neighbourhood we use *dom/wdeg* variable ordering [Boussemart *et al.*, 2004], ascending value ordering and a limit of 50 backtracks (a value found to be good in preliminary experiments). To find the first solution we replicate the method in §4.2 in Choco, except that the time limit is initially 1 second rather than 100 ms.

SNS, OscaR/CBLS and the LNS methods are randomised while Chuffed is deterministic. For each solver and instance we perform 10 runs each with a 10 minute time limit. Experiments were run on an Intel Xeon E5-2640 v4 at 2.40GHz with 20 cores (40 hyper-threads) with 20 processes run in parallel.

Table 3 summarises our results. SNS is extremely effective: this is indicated by the proliferation of '1' entries in the first column of SNS results, indicating that SNS found the best results (or tied) of any competing solver on those instances. Also, SNS was the only solver to successfully find a solution in every run (indicated by the absence of ∞'s in the second column). The two LNS variants are also effective, but SNS is more effective overall. SNS outperforms the LNS variants on all CVRP and SONET instances. On mean best score found, for CVRP the wins are by 6.7% (EB) and 19% (PG), while for SONET the wins are by 10% (EB) and 0.2% (PG). On PPP, SNS is 0.4% better than LNS EB, and 3.4% worse than LNS PG. On the MEB instances, SNS is 2.7% worse than LNS EB, and 4.5% better than LNS PG. Both LNS variants outperform SNS on the RackConf instances, although by only 0.2%.

The problem with the most structure in the variables, CVRP, is one where SNS performs particularly well. CVRP is specified with a single decision variable, but it has a nested type: a set of sequences of integers. SNS generates both lifted and direct neighbourhoods of sequences and it substantially outperforms all other solvers. Our results bear out our hypothesis: specifications can be exploited to automatically generate effective neighbourhoods for constraint-based local search.

6 Conclusion and Further Work

Structured Neighbourhood Search (SNS) exploits abstract structure in constraint specifications automatically to construct neighbourhoods that reflect that structure, and then performs local search. Our implementation outperforms variants of Large Neighbourhood Search (LNS) and Constraint-based Local Search (CBLS), especially on instances with complex abstract structure. This is particularly notable given

Instance	*opt*	SNS		OscaR/CBLS		Chuffed		LNS EB		LNS PG	
CVRP-1	290	**1**	1.002	1.048	1.166	1.021	1.021	**1**	1.001	**1**	**1**
CVRP-2	375	**1**	1.025	1.808	∞	2.216	2.216	1.101	1.147	1.104	1.255
CVRP-3	569	**1**	1.077	2.058	∞	2.162	2.162	1.104	1.181	1.336	1.434
CVRP-4	529	**1**	1.179	∞	∞	3.837	∞	1.129	1.34	1.499	1.683
CVRP-5	114	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
PPP-1	13	**1**	1.054	∞	∞	**1**	1.008	**1**	1.054	**1**	1.038
PPP-2	12	1.167	1.258	∞	∞	**1**	**1**	1.167	1.25	1.083	1.225
PPP-3	12	1.167	1.225	∞	∞	1.25	1.25	1.167	1.233	**1**	1.183
PPP-4	13	**1**	1.1	∞	∞	1.077	1.077	1.077	1.146	1.077	1.177
PPP-5	14	**1**	1.064	∞	∞	**1**	**1**	**1**	1.029	**1**	1.036
PPP-6	13	1.077	1.123	∞	∞	1.077	1.077	1.077	1.108	**1**	1.131
PPP-7	13	**1**	1.115	∞	∞	**1**	**1**	1.077	1.162	1.077	1.085
PPP-8	13	**1**	1.092	∞	∞	1.077	1.077	**1**	1.108	**1**	1.092
PPP-9	13	1.077	1.131	∞	∞	**1**	**1**	1.077	1.162	**1**	1.092
MEB-01	202	**1**	**1**	**1**	1.121	2.98	3.05	**1**	**1**	**1**	**1**
MEB-02	205	**1**	1.16	1.634	2.391	5.868	5.868	**1**	1.001	**1**	1.032
MEB-03	328	**1**	**1**	**1**	1.005	**1**	**1**	**1**	**1**	**1**	**1**
MEB-04	160	**1**	**1**	1.456	1.501	**1**	**1**	**1**	**1**	**1**	**1**
MEB-05	52	1.423	1.869	2.038	∞	6.808	6.808	**1**	1.396	1.538	1.998
MEB-06	206	**1**	**1**	1.005	1.013	2.51	2.552	**1**	**1**	**1**	1.001
MEB-07	542	**1**	1.076	1.083	1.607	2.616	2.616	1.017	1.05	1.015	1.06
MEB-08	581	**1**	1.039	1.093	1.308	3.036	3.146	**1**	1.019	1.048	1.102
MEB-09	344	**1**	1.251	1.23	1.368	2.712	2.712	1.067	1.151	1.067	1.414
MEB-10	397	**1**	1.021	1.06	1.597	2.499	2.499	**1**	1.008	**1**	1.012
MEB-11	466	**1**	1.088	1.182	1.223	2.622	2.681	1.036	1.134	1.137	1.182
MEB-12	264	1.057	1.203	1.455	1.659	6.242	6.245	**1**	1.297	1.277	1.596
MEB-13	435	**1**	1.087	1.039	1.147	1.766	1.766	**1**	**1**	**1**	1.054
SONET-1	59	**1**	1.027	1.068	1.224	3.407	3.414	1.051	1.093	1.136	1.269
SONET-2	115	**1**	1.058	1.13	1.421	6.417	6.417	1.07	1.233	1.009	1.067
SONET-3	89	**1**	1.035	1.112	1.307	4.551	4.551	1.112	1.175	1.022	1.093
SONET-4	122	**1**	1.089	1.082	1.234	5.549	5.557	1.213	1.287	1.074	1.136
SONET-5	167	**1**	1.169	1.054	1.314	5.491	5.491	1.18	1.283	1.024	1.066
SONET-6	172	**1**	1.181	1.052	1.324	5.122	5.122	1.215	1.392	1.029	1.117
SONET-7	211	1.19	1.49	**1**	1.305	5.744	5.744	1.341	∞	1.142	∞
SONET-8	195	1.292	1.525	**1**	1.276	7.959	7.959	1.19	∞	1.056	∞
RackConf-01	550	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
RackConf-02	1100	**1**	**1**	1.091	1.259	**1**	**1**	**1**	**1**	**1**	**1**
RackConf-03	1200	**1**	**1**	1.583	1.725	**1**	**1**	**1**	**1**	**1**	**1**
RackConf-04	1150	**1**	**1**	1.043	1.061	**1**	**1**	**1**	**1**	**1**	**1**
RackConf-05	987	1.001	1.002	1.048	1.074	1.02	1.02	**1**	1.001	1.001	1.001
RackConf-06	624	**1**	1.017	∞	∞	**1**	**1**	**1**	1.004	**1**	**1**
RackConf-07	664	1.011	1.016	∞	∞	1.018	1.018	**1**	1.006	**1**	**1**
RackConf-08	711	1.003	1.015	∞	∞	1.025	1.025	**1**	1.005	**1**	**1**
RackConf-09	753	1.005	1.009	∞	∞	1.04	1.041	**1**	1.005	**1**	**1**
RackConf-10	783	1.005	1.018	∞	∞	1.033	1.033	**1**	1.019	**1**	1.003
RackConf-11	584	**1**	1.003	1.082	1.12	1.051	1.052	**1**	1.014	**1**	1.002

Table 3: Column *opt* lists the best value found by any solver. For each algorithm, the first column gives the ratio of the best value found to *opt* (lower is better with 1 best possible); ∞ indicates that no run yielded a solution. The second column gives the ratio of the mean of the best values found at the end of each run to *opt* (lower is better); ∞ indicates at least one run during which no solution was found.

the automatic generation of neighbourhoods from a very high level specification of a constraint problem. Future research includes optimising SNS parameters, or investigating more efficient exploration of the automatically constructed neighbourhoods.

Acknowledgements

We thank Nguyen Dang for her assistance with the experiments, and EPSRC grants EP/P015638/1 and EP/P026842/1 for funding. Christopher Jefferson is supported by a Royal Society University Research Fellowship.

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.

[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.

[Benoist *et al.*, 2011] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. LocalSolver 1.x: a black-box local-search solver for 0-1 programming. *4OR*, 9(3):299–316, 2011.

[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.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[Chu, 2016] Geoffrey Chu. Chuffed, a lazy clause generation solver, 2016. Available from `https://github.com/chuffed/chuffed`.

[Cipriano *et al.*, 2013] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. A multi-paradigm tool for large neighborhood search. In *Hybrid Metaheuristics*, pages 389–414, 2013.

[Codognet and Diaz, 2001] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *SAGA*, LNCS 2264, pages 73–90. Springer, 2001.

[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] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints 13(3)*, pages 268–306, 2008.

[Gent *et al.*, 2006] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast, scalable, constraint solver. In *ECAI*, pages 98–102, 2006.

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

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, LNCS 4212, pages 282–293. Springer, 2006.

[Munos, 2014] Rémi Munos. From bandits to Monte-Carlo tree search: The optimistic principle applied to optimization and planning. *FTML*, 7(1):1–129, 2014.

[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.

[Pesant and Gendreau, 1999] Gilles Pesant and Michel Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 5(3):255–279, 1999.

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

[Rendl *et al.*, 2015] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. MiniSearch: A solver-independent meta-search language for MiniZinc. In *CP*, LNCS 9255, pages 376–392. Springer, 2015.

[Rendl, 2010] Andrea Rendl. *Effective Compilation of Constraint Models*. Ph.D. Thesis, Univ. of St Andrews, 2010.

[Ropke and Pisinger, 2006] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[Shaw, 1998] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, LNCS 1520, pages 417–431. Springer, 1998.

[Van Hentenryck and Michel, 2005] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. MIT Press, 2005.

[Van Hentenryck and Michel, 2007] Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. In *AAAI*, pages 273–278. AAAI Press, 2007.

[Van Hentenryck *et al.*, 1999] Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and Jean-Charles Régin. Constraint programming in OPL. In *PPDP*, LNCS 1702, pages 98–116, 1999.

[Ågren *et al.*, 2009] Magnus Ågren, Pierre Flener, and Justin Pearson. Revisiting constraint-directed search. *Information and Computation*, 207(3):438 – 457, 2009.