# The Open Stacks Problem
## An automated modelling case study

Ozgur Akgun, Ian Miguel, Christopher Jefferson
{ozgur,ianm,caj}@cs.st-andrews.ac.uk

School of Computer Science, University of St. Andrews, UK

**Abstract.** This paper presents a case study of automated modelling using Essence and Conjure. The problem we study is the open stacks problem[5]. We start with a naive problem specification and show how Conjure generates a selection of constraint programming models automatically. After observing the results, we modify the original problem specification to generate better models. Finally, we further improve the generated models by introducing a new representation for partial and injective function variables. The newly added representation is useful for not only this specific problem but also any other problem with a similar structure.

## 1 Introduction

This paper presents a case study of automated modelling using Conjure. Conjure is an automated modelling tool whose inputs are a high level problem specification given in Essence[2], and a collection of refinement rules. It performs problem class level transformations to generate multiple Essence′[4] models for the input problem specification.

Often referred to as the *modelling bottleneck*, correctly modelling a given problem is not a trivial task. Furthermore, producing a *good* model for a given problem is an active area of research. Even for constraint modelling experts, producing a good model remains an art rather than a science – we are far from fully understanding when a given model is better than another one.

In contrast to many existing languages for constraint modelling which only support decision variables of primitive types (boolean and integer), Essence provides complex types for decision variables and parameters to ease modelling. In order to tackle the challenge of producing good models, Conjure takes a rather unique approach. The applied transformations are extensible by design, to be able to capture new modelling idioms as they are discovered by human experts. Furthermore, the objective of the system is to generate all possible models using the available refinement rules. The output is not a single random model, instead it is a portfolio of models ready for further investigation.

The typical use case for a problem owner doesn't require any alterations to the refinement rules. The existing refinement rules are capable of refining all types and expressions found in Essence to generate valid models. Refinement

rule authoring is only required when a new variable representation or a new transformation is discovered.

The rest of the paper is structured as follows. We first give a precise description of the open stacks problem together with a simple example. In section 2, we give the architecture of the CONJURE system. Section 3 gives a precise problem specification and demonstrates how the system operates on it with the help of some example refinement rules. Section 4 builds on an observation specific to the problem at hand, and improves the problem specification. Finally, we compare the two specifications and conclude.

### 1.1 The problem

Taken from the Constraint Modelling Challenge 2005 proceedings[5]:

> A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customers order is started (i.e. the first product in the order is being made) a stack is created for that customer. [Each customer places exactly one order.] When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the maximum number of stacks that are in use simultaneously, i.e. the number of customer orders that are in simultaneous production, should be minimised.

For clarity, we introduce a simple example consisting of 5 products and 5 customers. In Fig 1, the demand matrix, $D$, is given. Every row in this matrix correspond to a single customer, and every column correspond to a single product. $D_{ij}$ is 1 iff $c_i$ placed an order for $p_j$.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $c_1$ | 1 | 1 | 0 | 1 | 0 |
| $c_2$ | 0 | 1 | 0 | 1 | 1 |
| $c_3$ | 0 | 0 | 1 | 1 | 0 |
| $c_4$ | 0 | 0 | 1 | 0 | 0 |
| $c_5$ | 0 | 0 | 1 | 0 | 0 |

**Fig. 1.** $D$, the demand matrix

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $c_1$ | 1 | 1 | 1 | 1 | 0 |
| $c_2$ | 0 | 1 | 1 | 1 | 1 |
| $c_3$ | 0 | 0 | 1 | 1 | 0 |
| $c_4$ | 0 | 0 | 1 | 0 | 0 |
| $c_5$ | 0 | 0 | 1 | 0 | 0 |

**Fig. 2.** $O$, the open stacks matrix

In order to calculate the number of stacks needed for this instance, we use an intermediate matrix of boolean variables, the open stacks matrix shown in Fig 2. $O_{ij}$ is 1 iff a stack for $c_i$ is open at the time of producing $p_j$.

Using this example and without permuting the order of production, at the time of producing $p_3$, all 5 stacks are needed. However, as can be seen in Fig 3 changing the order of production to $(p_1, p_2, p_4, p_5, p_3)$ at most 3 stacks are needed at any time.

| | $p_1$ | $p_2$ | $p_4$ | $p_5$ | $p_3$ |
|---|---|---|---|---|---|
| $c_1$ | 1 | 1 | 1 | 0 | 0 |
| $c_2$ | 0 | 1 | 1 | 1 | 0 |
| $c_3$ | 0 | 0 | 1 | 1 | 1 |
| $c_4$ | 0 | 0 | 0 | 0 | 1 |
| $c_5$ | 0 | 0 | 0 | 0 | 1 |

**Fig. 3.** The optimal solution using only 3 stacks

## 2 The architecture of Conjure

Conjure 1.0 is structured like a compiler. It operates by applying the given refinement rules to the input problem specification in several reentrant phases. The refinement rules are annotated by a precedence level; rules with higher precedence are always applied before the others. The rules residing at the same precedence level are applied simultaneously to generate multiple output models. Once a rule is applied, the process continues from the highest precedence level.

There are two main kinds of refinement rules, rules to select a concrete representation for an Essence type, and rules to transform Essence expressions.

The pipeline starts with parsing, validating the input, and type-checking. After these foundation phases, it prepares the input specification for, and performs, refinement, and does some housekeeping:

1. Parsing
2. Validation
   - *Are all identifiers defined?*
   - *Check consistency of declarations. e.g. a function variable cannot be declared both total and partial.*
3. Type Checking
4. Refinement
5. Model Presentation

Phases 1–3 are foundational, while Phase 5 aids perspicuity. Phase 4 is the core of the refinement process, and is briefly discussed below. It consists of multiple reentrant levels: following each rule application the process returns to Phase 4i) in case the result of the rule requires the attention of any of the other levels. We summarise Phase 4:

**4i) Partial Evaluation** Conjure 1.0 contains a partial evaluator for Essence. This not only simplifies the output models, but also saves the system from applying rules to expressions that can readily be evaluated.

**4ii) Representation Selection** Refinement of an abstract expression depends crucially on the representation of the abstract decision variables it involves. Hence, it is natural to select decision variable representations first. This also simplifies the generation of channelling constraints (Level iii) considerably. Typically *structural constraints* are added to the variables in the concrete representation to ensure that the abstract variable is properly represented.

**4iii) Auto-Channelling** When an abstract decision variable appears in multiple constraints, it can also have multiple representations in a single model (to suit each constraint), in which case *channelling* constraints [1] are necessary to maintain consistency among these different representations. Following [3], channelling constraints are generated simply by constraining the different representations of each abstract variable so that they represent the same abstract object. The resultant equality constraints are refined in the same way as any other constraint in the specification.

**4iv) Expression Refinement** Having decided on the representation of each abstract decision variable, it remains to refine the expressions that contain them.

In order to produce multiple models, refinement branches in two places in Phase 4: Representation Selection and Expression Refinement. Depending on the rules available in the rule base, each abstract decision variable and each expression can be refined in several different ways.

## 3 A *precise* problem description

In this section we give a precise problem description, first in English and then in ESSENCE.

> Given a number of customers, a number of products, and a demand relation between customers and products, find a permutation of products, such that if production is done in this order the number of stacks needed at any time is minimised. A stack is opened for every customer when the first product they demand is produced and it is closed when there are no more products for them to be produced.

The ESSENCE problem specification (Fig 4) is very close to the English description and should be readily understandable. The problem is parameterised over two integers and one 2 component relation. Three letting statements are used to bind names to commonly used domains. Notice, two of these names, *PRODUCT* and *TIMESLOT*, are actually bound to the same domain; different names are introduced to capture the different meanings and ease understanding in the rest of the problem specification.

The decision central to the problem is finding a permutation of the given products, hence the function variable named *timeof*. It is decorated with two attributes; *total* to ensure every value in the domain set is assigned to a value from the range set, and *injective* to ensure that the function preserves distinctness.

The function variable named *stackOpen*, mapping every customer to a pair of time slots, is used to mark the stack opening and closing times for every customer. The relation variable *isOpenStack* is only used to ease the calculation of *nbStacks*, the maximum number of stacks needed at any time slot.

First two constraints, lines 16 and 17 in Fig 4, constrain a stack for a customer to remain open for the duration of all the demand points for that customer. The

```
1  language Essence 2.0
2
3  given   nbProducts, nbCustomers: int(1..)
4  letting PRODUCT  be domain int(1..nbProducts),
5          TIMESLOT be domain int(1..nbProducts),
6          CUSTOMER be domain int(1..nbCustomers)
7  given   demand      : relation of (CUSTOMER * PRODUCT)
8  find    timeof      : function (total, injective) PRODUCT → TIMESLOT
9  find    stackOpen   : function (total) CUSTOMER → tuple (TIMESLOT,TIMESLOT)
10 find    isOpenStack : relation of (CUSTOMER * PRODUCT)
11 find    nbStacks    : PRODUCT
12
13 minimising nbStacks
14
15 such that
16     forall (c,p) : demand . stackOpen(c)[0] ≤ timeof(p),
17     forall (c,p) : demand . stackOpen(c)[1] ≥ timeof(p),
18
19     forall c : CUSTOMER . forall t : TIMESLOT .
20         isOpenStack(c,t) = (stackOpen(c)[0] ≤ t ∧
21                             stackOpen(c)[1] ≥ t),
22
23     forall t : TIMESLOT .
24         nbStacks ≥ |isOpenStack(_,t)|
```

**Fig. 4.** Problem specification in ESSENCE

universal quantification over *demand*, gives only those customer–product pairs for which a demand exists. The third constraint simply relates the function variable to the relation variable such that *isOpenStack(c,t)* is true iff a stack for customer $c$ is open at time slot $t$.

### 3.1 Refinement

The process of auto-modelling the given problem specification is explained in this section. There are 3 decision variables and 1 parameter requiring refinement. CONJURE rule-base contains 2 refinement options for relation variables, and 2 refinement options for function variables. Using all the existing refinement rules, at least 16 valid models can be generated; some with multiple representations for a single decision variable and with the appropriate channelling constraints in place. Here, we only give those refinement rules necessary to generate one of the resultant models, arguably the best one.

All CONJURE rules adhere to a single template:

```
<pattern> [⇝ <output>]*
          [where <guards>]
          [letting <local identifiers>]
```

A rule matches against `pattern`, producing one or more `outputs` provided the `guards` are satisfied. Local identifiers are used for concision and to identify new variables created by the refinement process.

First, we start by giving the necessary representation selection rules. A representation selection rule, matches with an ESSENCE type, and outputs 3 things: the name of the representation selected, the concrete representation in the form

of a valid ESSENCE type, and any structural constraints to be added when this rule is applied. Every output is preceded by a $\rightsquigarrow$ sign. The structural constraints component is optional, it can be omitted if the type refinement doesn't imply any new constraints.

```
function (total, injective) a → b ⇝ Function1DMatrix
                                  ⇝ matrix indexed by [a] of b
                                  ⇝ alldifferent(refn)
    where a :: int
```

**Fig. 5.** Representation selection rule for a total and injective function variable.

Fig 5 gives a representation selection rule for total and injective function variables. A decision variable with a matching type can be refined to a one-dimensional matrix with an *alldifferent* posed on it. Here, *refn* is a special operator which returns the newly created variable after applying this representation selection rule.

```
function (total) a → b ⇝ Function1DMatrix
                       ⇝ matrix indexed by [a] of b
    where a :: int
```

**Fig. 6.** Representation selection rule for a total function variable.

Fig 6 gives a representation selection rule for total function variables. Similar to Fig 5, a decision variable with a matching type can be refined to a one-dimensional matrix. Notice, we don't pose an *alldifferent* constraint in this case, since the function variable is not marked to be injective.

```
relation of (a * b) ⇝ Relation2D
                    ⇝ matrix indexed by [a,b] of bool
    where a :: int, b :: int
```

**Fig. 7.** Representation selection rule for a 2-component relation.

Fig 7 gives a representation selection rule for 2-component relation variables. Both the parameter *demand* and decision variable *isOpenStack* will use this representation during refinement.

Fig 8 gives a refinement rule which fires while refining the first two constraints, on lines 16 and 17 in Fig 4. The pattern on the left hand side of the $\rightsquigarrow$ sign, matches with a universal quantification over a 2-component matrix. `k` in the pattern, matches with the body of the quantification. The returned expression contains a guard in the form of an logical implication on `k`, ensuring k is only ever applied for values found in the relation *rel*.

There are some important operators used in this refinement rule:

`(::)` does type checking. It accepts two arguments, and returns *true* if the types of the parameters are same, and *false* otherwise. Each parameter can be an ESSENCE expression or an ESSENCE types.

`repr` returns the selected representation for an atomic expression. It can only be used in the *where* clause of a refinement rule and fails the application of the rule if a representation is not chosen for its argument.

```
forall (i,j) : rel . k
    ~> forall i : ri .
          forall j : rj .
             m[i,j] => k
    where   rel :: relation of (int,int),
            repr(rel) = Relation2D
    letting m be refn(rel),
            ri be indices(m)[0],
            rj be indices(m)[1]
```

**Fig. 8.** Refinement rule for quantification over a 2-component relation with *Relation2D* as the selected representation.

`refn` returns the concrete representation for an atomic expression.

`indices` returns the indexing domains of a matrix. It returns a tuple containing every indexing domain in their respective positions. Individual indices can be projected out of the tuple using the tuple indexing operator `[]`.

```
|rel(_,b)| ~> sum (i,j) : rel . (j = b) * rel(i,j)
```

**Fig. 9.** Horizontal refinement rule for cardinality of relation projection.

This rule given in Fig 9 is used while refining the last constraint, starting on line 23 in Fig 4. It is called a *horizontal* rule, because it doesn't refer to the representation of any decision variable using the *repr* operator, nor it requests the refinement of a decision variable using the *refn* operator. The existence of horizontal rules are very important to the scalability of the rules database. A vertical rule needs to be added for every newly added variable representation, whereas horizontal rules can be used independently of the chosen representation.

Notice, since horizontal rules do not do type refinement, the resultant expression needs further refinement rule applications. In this specific case, a rule very similar to that of Fig 8 will be applied to refine the relation variable to a matrix.

Given refinement rules are sufficient to produce the output model given in Fig 11. Although automatically generated from a problem specification, the model is very close to what a human modeller would write, once the modelling decisions are decided upon. Except the last constraint, starting on line 28, which would have been written as in Fig 10. This is due to using a horizontal rule to transform the cardinality constraint to a nested *sum*. Luckily, in this case and in many other cases where we consider the use of horizontal rules to be beneficial, the two forms are identical modulo instantiation. Any tool which instantiates the problem class model with given parameters will generate exactly the same constraints.

```
    forall t : TIMESLOT .
        nbStacks ≥ sum c : CUSTOMER . isOpenStack[c,t]
```

**Fig. 10.** The last constraint of Fig 11, slightly modified.

```
language ESSENCE' 1.0

given nbProducts, nbCustomers : int(1..)
letting PRODUCT  be domain int(1..nbProducts),
        TIMESLOT be domain int(1..nbProducts),
        CUSTOMER be domain int(1..nbCustomers)
given demand      : matrix indexed by [CUSTOMER,PRODUCT] of bool
find timeof       : matrix indexed by [PRODUCT] of TIMESLOT
find stackOpened  : matrix indexed by [CUSTOMER] of TIMESLOT
find stackClosed  : matrix indexed by [CUSTOMER] of TIMESLOT
find isOpenStack  : matrix indexed by [CUSTOMER,TIMESLOT] of bool
find nbStacks     : PRODUCT

minimising nbStacks

such that
    alldifferent(timeof),

    forall c : CUSTOMER . forall p : PRODUCT .
        demand[c,p] ⇒ stackOpened[c] ≤ timeof[p],

    forall c : CUSTOMER . forall p : PRODUCT .
        demand[c,p] ⇒ stackClosed[c] ≥ timeof[p],

    forall c : CUSTOMER . forall t : TIMESLOT .
        isOpenStack[c,t] = ((stackOpened[c] ≤ t) ∧ (t ≤ stackClosed[c])),

    forall t : TIMESLOT .
        nbStacks ≥ sum t2 : TIMESLOT . (
                        (t = t2) *
                        (sum c : CUSTOMER . isOpenStack[c,t])
                    )
```

**Fig. 11.** The auto-generated model for the problem specification given in Fig 4.

## 4 Better understanding the problem

As noted in [5], an important observation about the problem enables us to drastically shrink the problem size.

**Observation.** For every product $p$, if there exists another product $p'$ such that the set of customers demanding $p$ is a *subset* of the set of customers demanding $p'$, $p$ doesn't need to be considered while sequencing products.

For example, in the instance given at Fig 1, products $p_1$, $p_2$ and $p_5$ do not need to be considered while sequencing the rest of the products. Namely, only sequencing $p_3$ and $p_4$ and minimising the required number of stacks for these two products is enough to optimally solve the original problem.

The observation means we can preprocess the data file to remove such products, and use the problem specification we already have. Alternatively, we can encode the property in the problem specification; enabling us to leverage from the observation while still using Essence and staying at the problem class level.

In the light of this observation, instead of finding a total mapping from products to time-slots, we need to search for a partial mapping. To accomplish this, we simply modify the attributes of *timeof* to include *partial* instead of *total*.

```
find timeof : function (partial, injective) PRODUCT → TIMESLOT
```

*timeof* being partial function means some products will be assigned to time-slots and others won't. Since the function is still *injective*, it preserves distinctness through assigned time-slots.

Two constraints are needed to statically compute those products which need sequencing.

```
forall p1 : PRODUCT . (
   (exists p2 : PRODUCT . (p1 ≠ p2) ∧ (demand(_,p1) ⊆ demand(_,p2)))
   ⇒ (p1 ∉ defined(timeof))
),

forall p1 : PRODUCT . (
   (forall p2 : PRODUCT . (p1 ≠ p2) ⇒ !(demand(_,p1) ⊆ demand(_,p2)))
   ⇒ (p1 ∈ defined(timeof))
),
```

The operator `defined` works on function variables and returns the set of values the function is defined on.

Only the second and third constraints from the original problem specification (Fig 4) need to be modified slightly.

```
forall (c,p) : demand .
   p ∈ defined(timeof) ⇒ stackOpened(c) ≤ timeof(p),

forall (c,p) : demand .
   p ∈ defined(timeof) ⇒ stackClosed(c) ≥ timeof(p),
```

Other constraints remain unchanged.

The changes made on the problem specification arise from a better understanding of the problem. Although this may be perceived as pushing the modelling bottleneck up to the ESSENCE level, it should be noted that the modifications are done at problem specification level. Modelling decisions, such as how to model a partial and injective function variable, do not need to be considered. We improve the problem specification merely by changing how a decision variable is declared.

### 4.1 Refinement

CONJURE can already refine the modified problem description to valid constraint models. In order to be as generic as possible, the existing refinement uses two 1-dimensional matrices to represent a partial and injective function variable. Using this representation (Fig 12), the constraint to pose distinctness is highly inefficient.

In the specific case of a function variable mapping integers to integers, we can do better. Instead of introducing a boolean variable for every possible mapping, we can introduce a dummy value in the mapped domain, and use *alldifferent_except* to ensure distinct assignments (Fig 13).

```
function (partial, injective) a → b
    ⇝ PartialFunction1DMatrix
    ⇝ matrix indexed by [a] of tuple (bool,b)
    ⇝ forall i,j : a . (
          ((refn[i][0] = true) ∧ (refn[j][0] = true))
          ⇒ (refn[i][1] ≠ refn[j][1])
      )
    where a :: int
```

**Fig. 12.** Generic representation for partial and injective function variables.

```
function (partial, injective) a → int(1..b)
    ⇝ PartialFunctionIntInt
    ⇝ matrix indexed by [a] of int(0..b)
    ⇝ alldifferent_except(refn,0)
    where a :: int
```

**Fig. 13.** A specific representation for function variable mapping integers to integers.

This is a very specific variable representation, yet it occurs very often. In the process of modelling the open stacks problem, this specific case is discovered and proved to be highly efficient. Future problems with a similar structure will benefit from it for free.

The refinement rule in Fig 14 is needed to fully refine the modified problem specification using the newly added representation.

```
e ∈ defined(fn) ⇝ refn(fn)[e] > 0
    where repr(fn) = PartialFunctionIntInt
```

**Fig. 14.** A new refinement rule is necessary to fully refine the problem specification.

The result of refining the modified problem specification is given in Fig 15.

## 5 Experimental results

We ran experiments on the instances provided by [5]. Search node counts and search times are recorded for over 700 problem instances for both the original and the improved model.

The plots given in Fig 16 and Fig 17 are comparing the two models by the number of search nodes and by actual search time respectively. The instances are ordered by the amount of gain. Both vertical axes are log-scaled.

Comparing constraint models is not a trivial task. Given two problem class models, one can be better than the other for some parameter instantiations and worse for some others.

Number of search nodes used by the underlying solver is one possible measure for comparing two constraint models. It gives a good understanding as long as

```
language ESSENCE' 1.0

given nbProducts : int(1..)
letting PRODUCT be domain int(1..nbProducts)
letting TIMESLOT be domain int(1..nbProducts)

given nbOrders : int(1..)
letting CUSTOMER be domain int(1..nbOrders)

given demand      : matrix indexed by [CUSTOMER,PRODUCT] of bool
find timeof       : matrix indexed by [PRODUCT] of int(0..nbProducts)
find stackOpened  : matrix indexed by [CUSTOMER] of TIMESLOT
find stackClosed  : matrix indexed by [CUSTOMER] of TIMESLOT
find isOpenStack  : matrix indexed by [CUSTOMER,TIMESLOT] of bool
find nbStacks     : PRODUCT

minimising nbStacks

such that
    alldifferent_except(timeof,0),

    forall p1 : PRODUCT . (
        (exists p2 : PRODUCT . (
            (p1 ≠ p2) ∧
            (forall c : CUSTOMER . demand[c,p1] ≤ demand[c,p2]))
        ) ⇒ (timeof[p1] = 0)
    ),

    forall p1 : PRODUCT . (
        (forall p2 : PRODUCT . (
            (p1 ≠ p2) ⇒
            !(forall c : CUSTOMER . demand[c,p1] ≤ demand[c,p2]))
        ) ⇒ (timeof[p1] > 0)
    ),

    forall c : CUSTOMER . forall p : PRODUCT .
         (timeof[p] > 0 ∧ demand[c,p]) ⇒ stackOpened[c] ≤ timeof[p],

    forall c : CUSTOMER . forall p : PRODUCT .
        (timeof[p] > 0 ∧ demand[c,p]) ⇒ stackClosed[c] ≥ timeof[p],

    forall c : CUSTOMER . forall t : TIMESLOT .
        isOpenStack[c,t] = ((stackOpened[c] ≤ t) ∧ (t ≤ stackClosed[c])),

    forall t : TIMESLOT .
        nbStacks ≥ (sum c : CUSTOMER . isOpenStack[c,t])
```

**Fig. 15.** An improved model.

the propagation levels of the used constraints are similar. Some models may take far more search nodes to solve, yet can be solved faster.

In our experimental results, the fact that the *better* model runs faster than the *normal* model, as long as it needs fewer search nodes demonstrates a clear gain. Using a *partial* function variable instead of a *total* function variable not only saves from the number of search nodes, but also saves from the actual time spent during search.

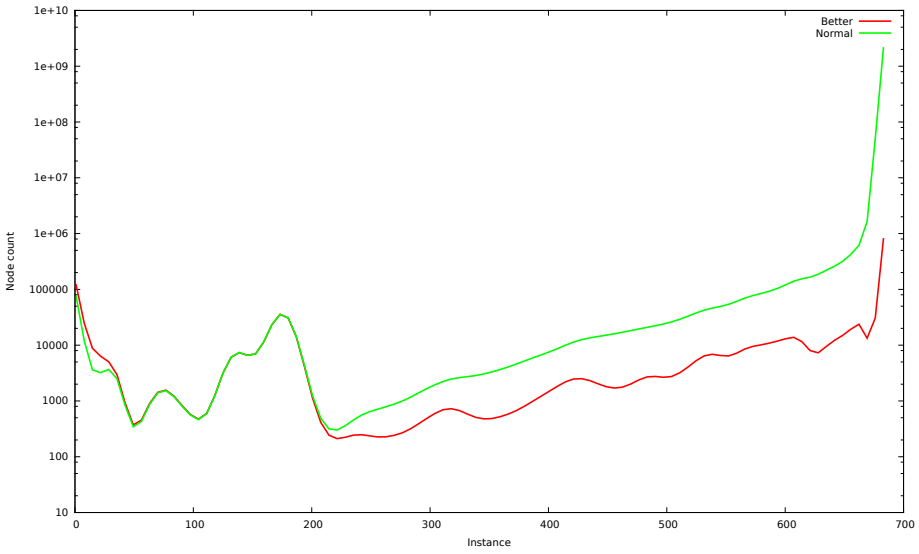


**Fig. 16.** Search nodes

## 6 Conclusion and Future work

We presented a use case for Conjure using the open stacks problem as an example. This exercise allowed us to discover a new and specific variable representation. The new refinement rules are added to the rules database; future refinements will benefit from these rules without any further work.

Conjure generates multiple models for a given problem specification. We didn't discuss how we select a good model out of the generated selection of models in this paper. This is partly because it was out of the focus of this paper and partly because we didn't develop any automated model selection techniques yet.

Now that Conjure can produce multiple valid constraint models for a given problem specification and the refinement language is mature enough to encode new transformations without modifying the internals, our aim is to study model

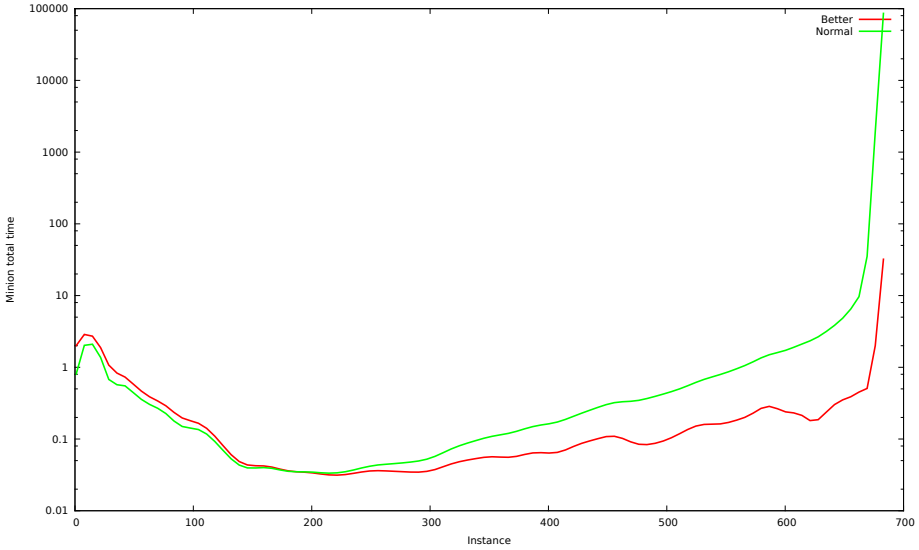


**Fig. 17.** Search times

selection; fully automated or assisted. We are also planning to investigate other problems to capture modelling idioms and enrich our refinement rules database.

### Acknowledgements

Ozgur Akgun is supported be a Scottish Informatics and Computer Science Alliance (SICSA) prize studentship. This research is supported by UK EPSRC grant no EP/H004092/1.

## References

1. B.M.W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
2. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints 13(3)*, pages 268–306, 2008.
3. Bernadette Martínez Hernández and Alan M. Frisch. The automatic generation of redundant representations and channelling constraints. In *Trends in Constraint Programming*, chapter 8, pages 163–182. ISTE, May 2007.
4. Andrea Rendl. *Thesis: Effective Compilation of Constraint Models.* PhD thesis, University of St. Andrews, 2010.
5. Barbara M. Smith and Ian P. Gent. Constraint modelling challenge, 2005. In conjunction with The Fifth Workshop on Modelling and Solving Problems with Constraints Held at IJCAI 2005, Edinburgh, Scotland, 31 July, 2005.