Instance Generation for Constraint Model Selection

Bilal Syed Hussain, Ian Miguel, and Ian Gent
bh246,ijm,ipg@st-andrews.ac.uk

University of St Andrews

Abstract Constraint modelling is widely considered the bottleneck to the adoption of constraint programming. The Conjure automated modelling system addresses this by allowing the user to write in the high level specification language Essence. This allows the user to use familiar mathematical notion such as sets & relations which can be arbitrarily nested. Currently Conjure can produce many alternative models, but to select between the models, instance data that can discriminate between the models has to be provided. The main contribution of this paper, is the automatic generation of instance data from an Essence specification.

1 Introduction

The formulating of effective models in terms of a constraint solver input language is the *modelling bottleneck* [8], which impedes the widespread adoption of constraint programming. Since there are many possible models of a problem and the model chosen can drastically affect runtime performance it can be very difficult for a novice to formulate an effective (or even correct) model from the many possible options. Automated constraint modelling is therefore a way of overcoming this difficulty. The Conjure automated modelling system [2] employes a refinement based approach, where the user writes his/her problem in the abstract specification language Essence [3]. Essence supports common mathematical notation such as functions, relations and sets which can be a arbitrarily nested, such as set of sets of functions.

Conjure is able to produce the kernels of constraint models, and through the use of racing can select the best models, if given appropriate parameter instance data [1]. This paper focuses on automatic parameter generation, which will create problem instance data useful to discriminate among the available models. The instance data is generated automatically from the specification through a guided exploration of the problem parameter space. In this paper, we demonstrate our approach the 'bias parameter generation method' on a small number of typical CSP problem classes. Our method has the added benefit that the results of the evaluation of the each generated instance can be reused in the racing process from Akgun et al [1].

Our work is related to automated parameter tuning [6] where the task is to find the configuration which gives the optimal performance for an algorithm A with parameters $p_1 ... p_k$. Here, methods such as F-Race use a training set of instances on which to perform the tests. In contrast, our aim is to identify instances that can be used to choose among constraint models.

2 The Importance of Choosing the Right model

Throughout this paper we consider three single parameter problem classes taken from CSPlib [4]: namely N-Queens, Traffic Lights [5] and Low Autocorrelation Binary Sequences(LABS) [7]. We focus on single parameter problems for simplicity and since for the chosen problem classes, are solvable for most instances.

To show the importance of choosing the correct model, the figures below show the runtime performance of finding an optimal model in these problem classes with various parameters. Each different point type in the plots represents a different model.

For the N-Queens problem (109 models, Figure 1), there is a single best model. The only model that could solve $n = 22$ and $n = 29$. These two parameters are discriminating parameters we want any new method to generate.

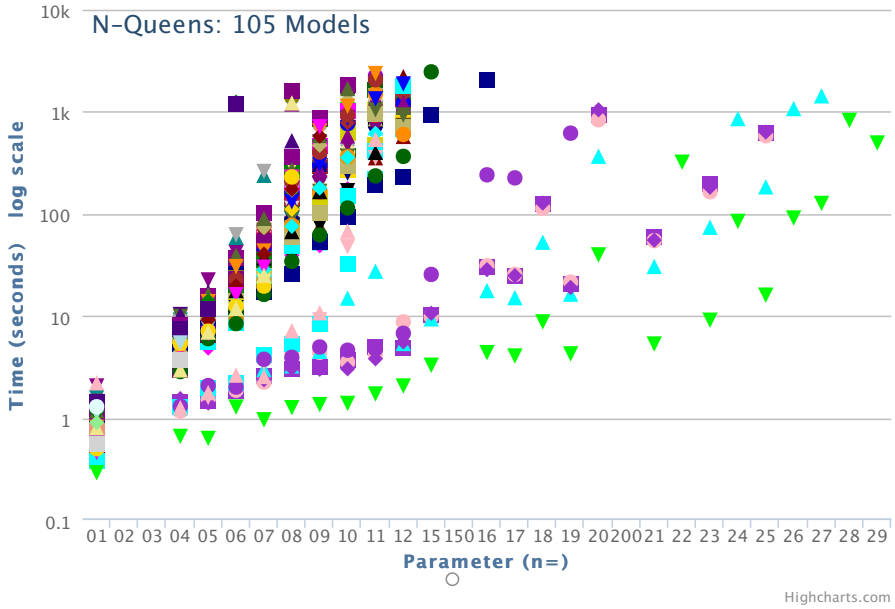

Figure 1. N-Queens with timeout 1500 seconds

For LABS (30 models, Figure 2) there are only three models that can solve $n > 23$ optimally, which shows that $n = 24$ is discriminating.

For Traffic Lights (16 models, Figure 3), there are two models that can solve large values of n, that is for parameters where $n > 6$, can pick two models from the original six.

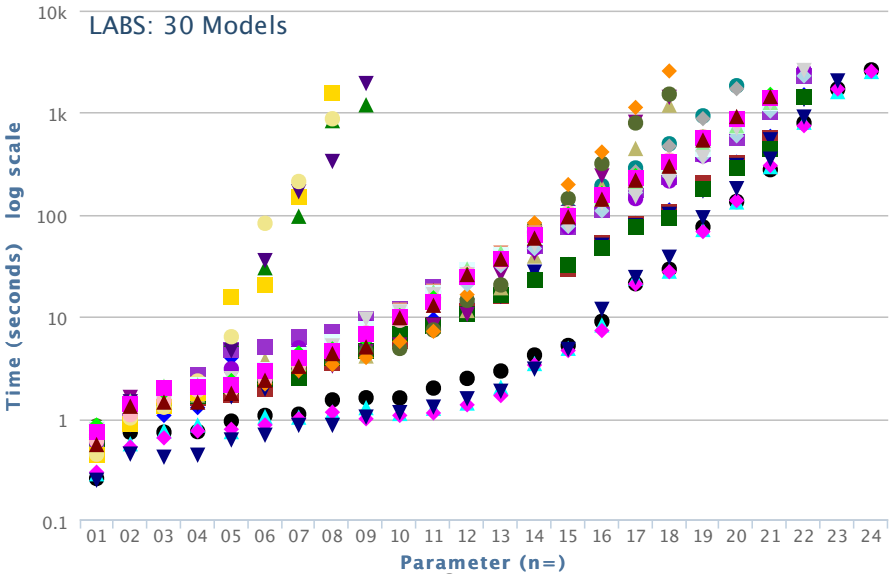

Figure 2. LABS with timeout 2600 seconds

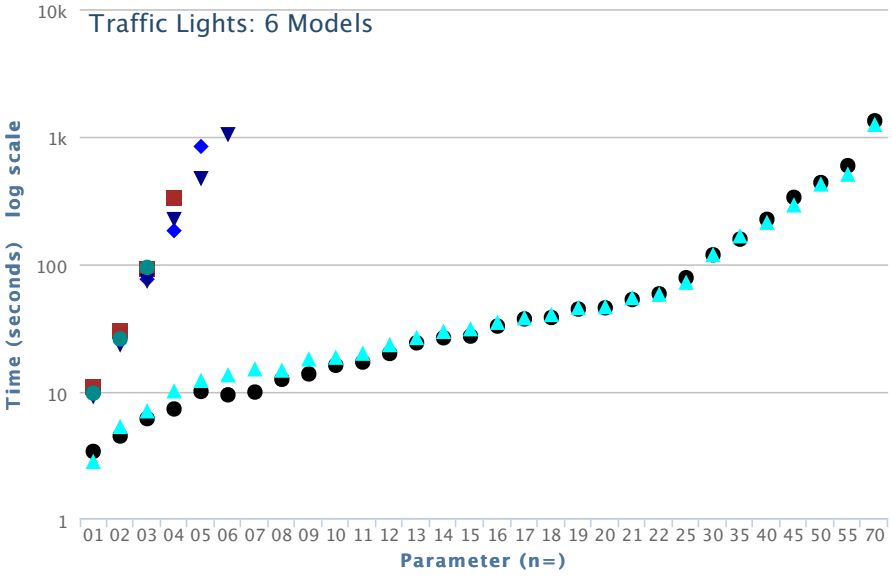

Figure 3. Traffic Lights with timeout 600 seconds

3 The Bias Parameter Generation Method

The aim is to generate parameters that can discriminate between the models that Conjure can produce. The criteria for the generated parameters are that within a set timelimit (i) at least one model can be solved or proven unsolvable, and (ii) not all models can be solved or can be proven unsolvable. If the user had the luxury to run every parameter, they of course would be able to find the parameters that discriminate. Since this is not feasible in general, one approximation is to use a binary search approach as shown below. This assumes the problem is monotonically increasing with respect to the size of the parameter.

```
given  E models, upper bound u, lower bound d, timeout t seconds
    while d <= u
        m <- (d + u) /2
        run parameter m on each model
        r <- number of models solved using m within t seconds
        if r > 0
            d <- m + 1
            if r < |E|
                store the param
        else
            u <- m -1
```

While this is faster than running every parameter, it places equal weight across the parameter space. For many problems there is a point where instances become harder; having more parameters close to this point would be useful for selection purposes.

The proposed improvement is to bias the jump to next parameter in the above algorithm by the ratio of number of models solved divided by the total number of models which allows the algorithm to focus in on the part of parameter space that would discriminate between the models.

```
given  E models, upper bound u, lower bound d, timeout t seconds
    while d <= u
        m <- (d + u) /2
        run parameter m on each model
        r <- number of models solved using m within t seconds
        if r > 0 and r < |E|
            d <- (d  +  (m - 1) * r/|E|) + 1
            if r < |E|
                store the param
        else
            u <- (u - (u - m ) * r/|E|) - 1
```

4 Experimental Evaluation

To test this method of parameter generation, the method was run on three problem classes namely N-queens, Traffic Lights and Low Autocorrelation Binary Sequences. The experiments were performed on a 32 core AMD Opteron 6272 at 2.1GHz.

This was compared with exhaustively running parameters on every model to verify that method did not miss any parameter that distinguish the models.

The table below shows the number of models for each problem, the number of models that could solve the most discriminating parameter from the full run and whether the bias method was able to produce a parameter that discriminates the models in the same way.

For each of the problem classes, the bias method found the number of models on the most discriminating parameter(s). The most discriminating parameter(s) is the one (set) that produces the lowest non-zero number of models that solve the problem. For N-Queens a single best model was found from 109 candidates. For Traffic Lights, two best models were found from 6, and for LABS three best models were found from 30.

Problem	Models	n	Number of Models on most discriminating param	Bias method
N-Queens	109	1..80	1	Yes
Traffic Lights	6	1..100	2	Yes
LABS	30	1..100	3	Yes

4.1 N-Queens

The bias method returns the following using a timeout of 1500 seconds and $n = 1...80$:

- No models can be solved for $n = 40$ so try $n = 20$
- 5 models can be solved for $n = 20$ so try $n = 21$
- 5 models can be solved for $n = 21$ so try $n = 22$
- Only 1 model can be solved for $n = 22$

The resulting parameters $n = 20$, $n = 21$ and $n = 22$ are very discriminating as shown in Figure 1. $n = 22$ can select the best model from 105 different models. The bias method is also significantly faster, 5.5 hours compared with 50 hours for exhaustively running the parameters. As compared to not using a bias which takes 6.5 hours to find the other discriminating param $n = 28$

4.2 Traffic Lights

The bias method returns the following using a timeout of 600 seconds and using $n = 1...100$:

- 2 models can be solved for $n = 50$ so try $n = 59$
- 2 models can be solved for $n = \{59, 66, 65\}$ so try $n = 71$
- no model can be solved for $n = 71$ so try $n = 64$
- 2 models can be solved for $n = \{64, 67\}$ so try $n = 70$
- no model can be solved for $n = 70$ so try $n = 66$,
- 2 models can be solved for $n = \{63, 65\}$

The resulting parameters can discriminate the two best models from the 6 starting models, The bias method finishes in 3 hours as compared with 5 hours for exhaustively running all parameters. Not using a bias results in a subset of the above parameters and is slightly faster taking just under 3 hours.

4.3 Low Autocorrelation Binary Sequences

The bias method returns the following using a timeout of 2000 seconds and using $n = 1...100$:

- no models can be solved optimally for $n = 50$ so try $n = 37$
- no models can be solved optimally for $n = 37$ so try $n = 28$
- 20 models can be solved optimally for $n = 21$ so try $n = 24$
- 3 models can be solved optimally for $n = 24$ so try $n = 25$
- no models can be solved optimally for $n = 25$ so try $n = 23$
- 8 models can be solved optimally for $n = 23$

The resulting parameters discriminate the best three models from the original 30. The bias method runs in 4 hours as compared with the 17 hours for exhaustively running all parameters. As compared to not using a bias it finds $n = 24$ in 4 hours as well. The difference being that the parameters are more spread out.

5 Conclusion & Future Work

This paper takes a first step towards automated instance generation for constraint model selection. Future work will include extending the bias parameter generation method to multi-parameter problems and integrating it into the racing approach of Akgun et al. [1].

Acknowledgements: Bilal Syed Hussain is supported by an EPSRC scholarship.

References

1. Akgun, O., Frisch, A.M., Hussain, B.S., Gent, I.P., Jefferson, C.A., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in conjure. In: CP (2013)
2. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAAI-11: Twenty-Fifth Conference on Artificial Intelligence (2011)
3. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: Proc. of the IJCAI 2005. pp. 109–116 (2005)
4. Hnich, B., Miguel, I., Gent, I.P., Walsh, T.: CSPLib: a problem library for constraints, http://csplib.org/
5. Hower, W.: Revisiting global constraint satisfaction. Information Processing Letters 66 (1998)
6. Hutter, F.: Automated Configuration of Algorithms for Solving Hard Computational Problems. Ph.D. thesis, University of British Columbia, Vancouver, Canada (October 2009)
7. Mertens, S.: Exhaustive search for low-autocorrelation binary sequences. J. Phys. A 29, L473–L481 (1996)
8. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: Principles and Practice of Constraint Programming - CP 2004. pp. 5–8 (2004)