

---

# Estimating Properties of Constraint Solution using Regression Trees

---

David DeVault

DDEVALT@CS.RUTGERS.EDU

Rutgers Computer Science, 110 Frelinghuysen Road, Piscataway, NJ 08854 USA

## Abstract

Formulating computational problems in a constraint satisfaction framework is often very attractive, as it allows a clear, declarative specification of solution properties, and offers the potential to exploit preexisting, general-purpose, highly-optimized constraint satisfaction algorithms. Yet most constraint satisfaction algorithms assume that constraint solutions can be efficiently tabulated, and that variable domains can be specified extensionally, in advance of problem solution. In many practical applications these assumptions cannot be met, necessitating the manual specification of an application-specific search strategy for constraint satisfaction. In this paper, I explore the use of regression tree learning to estimate the number of solutions a constraint will have, and the cost of finding them, from shallow constraint and variable assignment features. Reliable estimates of these factors may enable the use of principled cost-sensitive search techniques, with less manual intervention, alleviating much of the burden of exploiting constraint satisfaction for practical applications.

## 1. Introduction

Formulating computational problems in a constraint satisfaction framework is often very attractive, as it allows a clear, declarative specification of solution properties, and the potential to exploit preexisting, general-purpose, highly-optimized constraint satisfaction algorithms; see Dechter (2003). Yet most constraint satisfaction algorithms assume that constraint solutions can be efficiently tabulated, and that variable domains can be specified extensionally, in advance of problem solution. In many practical applications these assumptions cannot be met. For example, Jónsson & Frank (2000) utilize constraint satisfaction to specify planning problems for spacecraft. Their spacecraft need to be able to reason with arithmetic constraints on real

variables (whose domains cannot be enumerated extensionally) and with other constraints on acceptable plans whose solution complexity precludes the tabulation of all solutions in advance. Similarly, Chopra et al. (1996) exploit constraint satisfaction in a computer vision application, where constraints capture spatial relations between image features that are too expensive to be exhaustively tabulated in advance of constraint satisfaction. While these authors have suggested algorithms for computing arc consistency in networks of expensive constraints, it does not appear that a principled and practical approach to tabulating full solutions to such networks has been developed.<sup>1</sup>

As a step in this direction, in this paper I explore the use of regression tree learning to estimate the number of solutions a constraint will have, and the cost of finding these solutions, from shallow features of constraints and provisional variable assignments. If reliable estimates of these factors can be attained, it may enable the use of principled, cost-sensitive search techniques, with less manual intervention, alleviating much of the burden of exploiting constraint satisfaction for practical applications.

## 2. FIGLET

I approach this problem in the context of a figure-drawing application called FIGLET; see DeVault & Stone (2003) for details. FIGLET answers English-language instructions such as *Make the face bigger* by manipulating an evolving graphical figure. FIGLET identifies a user's intention by solving a constraint network. For example, FIGLET assigns the following constraints to *Make the face bigger*:

- (1) a.  $simple(A) \wedge target(A, X) \wedge$   
 $holds(now, fits\_plan(A)) \wedge$   
 $holds(result(A, now), visible(X)) \wedge$
- b.  $number(X, 1) \wedge holds(now, type(X, face)) \wedge$

---

<sup>1</sup>Although see Rathod et al. (2002) for some initial work on the use of cost-sensitive heuristics for computing full solutions.

- c.  $holds(now, size(X, SO)) \wedge$   
 $region(R, bigger\_than, SO) \wedge$   
 $in\_region(SN, R) \wedge$   
 $holds(result(A, now), size(X, SN)).$

Generally, all solutions to such constraints cannot be tabulated in advance. For example, a constraint of the form  $holds(RefSituation, size(RefObject, S))$  is satisfied when some object  $RefObject$  has size  $S$  in some reference situation  $RefSituation$ . The reference situation may be hypothetical, as in the last constraint of (1c), where the reference situation is  $result(A, now)$ , i.e. the situation resulting from carrying out action  $A$ . There are enough possible actions and resulting situations, and reasoning about these situations requires enough time, that enumerating all the possibly relevant solutions to constraints of this form is prohibitively expensive.

In our current system, it would be possible to enumerate the possible values of discrete-valued variables like  $A$  and  $X$  in constraints like  $holds(result(A, now), visible(X))$ . However, we hope to avoid doing so, in order to decrease the amount of work required to maintain a functioning constraint satisfaction algorithm as we scale up FIGLET’s capabilities.

### 3. Method

The assumption underlying the work presented here is that, at a given point in searching for a solution to an expensive constraint network, we will have satisfied certain constraints, producing a set of candidate variable assignments, and we will have a set  $C$  of constraints left to be solved. The task is to decide which constraint in  $C$  to solve next.

Let  $C = \{ \langle p_1, a_1 \rangle, \dots, \langle p_n, a_n \rangle \}$  be a set of  $\langle \text{predicate}, \text{arity} \rangle$  pairs describing the set of constraints left to be solved. Let  $v_c$  denote a partial assignment of values to the  $(a_i)$  variable arguments of constraint  $c = \langle p, a \rangle$ , where  $c \in C$ . Let  $c[v_c]$  denote the constraint  $c$  under the variable assignment  $v_c$ . Each solution to  $c[v_c]$  is a new variable assignment extending  $v_c$ . In this project, I explored the estimation of

$$(2) SOL(c, v_c) = \text{number of solutions to } c[v_c]$$

and

$$(3) TIME(c, v_c) = \frac{\text{time required to find}}{\text{all solutions to } c[v_c]}.$$

I used regression trees to estimate this numeric output.

Regression trees are analogous to decision trees, except they produce numeric output rather than a classification at each leaf. I implemented the CART regression tree learning algorithm (Breiman et al., 1984) to learn two regression trees, one for  $SOL(c, v_c)$ , and one for  $TIME(c, v_c)$ . CART essentially functions like ID3, except it uses the variance in the output values for training data at each tree node, rather than the entropy with respect to a classification scheme, as a measure of purity. At each node, it selects the attribute that results in the lowest average variance in the training data across the child nodes. Using the average variance for attribute selection biases it toward the early selection of attributes which sort the training data into groups associated with tighter distributions of numeric output.

Based on my experience in hand-tuning a constraint satisfaction strategy for FIGLET, I chose the following input features to describe  $c = \langle p, a \rangle$  and  $v_c$ :

- (4) a. Predicate: the constraint predicate symbol  $p$ . Predicate  $\in \{holds, target, simple, number, region, in\_region\}$ .
- b. Arity: the constraint arity  $a$ . Arity  $\in \{1, 2, 3\}$ .
- c. For each of three possible arguments, which I label  $X1$ ,  $X2$ , and  $X3$ , its status under  $v_c$ :
  - $S$ : current status of this argument.  $S \in \{bound, unbound, n/a\}$ .  $S = n/a$  if predicate lacks this argument.
  - $F$ : functor of term argument is bound to.  $F \in \{now, visible, list, action, fits\_plan, result, 1, type, shape, multiple, size, shorter\_than, n/a\}$ .  $F = n/a$  if argument is not bound in  $v_c$ .
  - $V$ : number of top-level unbound variables in the term this argument is bound to.  $V \in \{0, 1, 2, 3\}$ .<sup>2</sup>

I collected data as FIGLET interpreted the sequence of utterances:  $\langle Make\ a\ mouth, Make\ the\ mouth\ a\ rectangle, Add\ the\ eyes, Draw\ a\ head, Flatten\ the\ head \rangle$ . I had FIGLET make random choices about the order in which to solve constraints, which resulted in data for a variety of constraint solution situations. An example of the resulting data, for solution of the constraint  $holds(result(make\_visible([smileyface1]), now), visible([eyes4], 1.0))$ , is:

$\langle Pred=holds, Arity=2,$   
 $X1S=bound, X1F=result, X1V=0,$

<sup>2</sup>Variables in further embedded terms were not counted.

X2S=bound,X2F=visible,X2V=0,  
X3S=n/a,X3F=n/a,X3V=0,  
SOL=1,TIME=30ms>.

## 4. Results and Discussion

The trees learned by the CART algorithm, in each case using 90% of the data as a training set, and the other 10% as a test set, are illustrated in Figure 1. Notice that the overall standard deviation (STD) in the training data for  $SOL(c, v_c)$  is 13.02 solutions. The regression tree estimating  $SOL(c, v_c)$  manages to capture most of the deviation in the number of constraint solutions, as indicated by the overall root mean squared error (RMSE) of 2.20 solutions. Furthermore, looking at the RMSE associated with the individual rules, it is clear that this regression tree is highly accurate in most cases, with a RMSE of less than 1 solution. The overall error is mostly due to a few types of constraints, including for example those matching rules S3 and S4. This is understandable. As can be seen in the indicated examples, S3 and S4 correspond to visibility constraints, which happen to have one solution for every subset of atomic objects visible on the screen in some situation. But the attributes do not record any information about how many atomic objects are visible on the screen. Since the number of solutions to these constraints grows exponentially with the number of visible atomic objects (which ranges from 0 to 7) it is reasonable to expect this regression tree to be unable to make an accurate prediction of the number of solutions to these constraints.

Looking at the regression tree estimating  $TIME(c, v_c)$ , we find a STD in the training data of 54.87 milliseconds. The regression tree captures considerably less of the deviation in this data, with a RMSE of 43.42 milliseconds. However, looking at the RMS errors broken down according to the individual rules, we can again see that the RMS error is actually quite low in most cases (less than a few milliseconds, compared to a STD in the training data of 54.87 milliseconds), with the majority of the error associated with just a few rules. For example, visibility constraints are again problematic (rules T16-T19), for similar reasons, with RMS errors in the 10-430 millisecond range.

A 10-fold cross-validation with these training data revealed a RMSE in the SOL estimate of 10.91 solutions, and a RMSE in the TIME estimate of 38.68 milliseconds. Note that the 10-fold RMS error of 10.91 for SOL is considerably higher than the overall RMS error of 2.20 solutions indicated in Figure 1. This is due to sparse data. There are several rules in these trees

for which there is only one data point which happens to be an outlier compared to most of the other data. For example, see the training data statistics for rules S22 and T11 in Figure 1. In the 10-fold analyses, for most folds, the overall RMS error is comparable to that indicated in the figure. But when such examples end up in the test data but not the training data, it has a considerable effect on the overall RMS error.

Overall, these results indicate that, for many types of FIGLET constraints, it is possible, using only relatively shallow features such as those I have used in this work, to make reliable predictions of the number of solutions and the time it will take to find them. For a few cases, it appears that deeper features are required before comparable reliability can be achieved. However, even this somewhat unreliable information may be sufficient to carry out an effective, cost-sensitive constraint satisfaction search. Constraints that have high RMS errors here also tend to have large mean values for SOL and TIME. It is likely that a cost-sensitive search would opt to solve other constraints first for this reason, so that the relative unreliability in the available estimates would not derail the algorithm. In future work, I will implement a cost-sensitive search algorithm and explore this issue further.

## References

- Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees*. Wadsworth Int. Group.
- Chopra, R., Srihari, R., & Ralston, A. (1996). *Expensive Constraints and Hyper-Arc Consistency*. CONSTRAINTS-96, Workshop on Constraint Satisfaction Techniques at Florida AI Research Symposium (FLAIRS96).
- Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.
- DeVault, D. & Stone, M. (2003). *Domain Inference in Incremental Interpretation* Fourth Workshop on Inference in Computational Semantics (ICoS-4), Nancy, France, September, 2003.
- Jónsson, A. & Frank, J. (2000). *A Framework for Dynamic Constraint Reasoning using Procedural Constraints*. European Artificial Intelligence Conference, 2000.
- Rathod, P., desJardins, M., & Sansare, S. (2002). *Interactive, Incremental Scheduling for Virtual Telescopes in Education*. Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space, Houston, Texas, October 27-29, 2002.

Number of constraint solutions (SOL)							
ID	Rule	Example	Training Data			Test Data	
			N	Mean	STD	N	RMSE
S1	X2V=1 ^ X2F=fits_plan	holds(.,fits_plan(A))	79	34.87	0.80	6	0.13
S2	X2V=1 ^ X2F=shape	holds(.,shape(X,rectangle))	1	4.0	0.0	4	0.47
S3	X2V=1 ^ X2F=visible ^ X1V=1	holds(result(A,now),visible(X))	2	309.0	48.0	45	11.46
S4	X2V=1 ^ X2F=visible ^ X1V=0 ^ X1F=now	holds(now,visible(X))	18	23.22	35.29	4	26.50
S5	X2V=0 ^ X2F=fits_plan	holds(.,fits_plan(action5))	544	0.87	2.49	55	4.63
S6	X2V=0 ^ X2F=1	number(.,1)	50	1.0	0.0	5	0.0
S7	X2V=0 ^ X2F=n/a ^ X1V=3	simple(action2(A,B,C))	43	0.33	0.47	3	0.33
S8	X2V=0 ^ X2F=n/a ^ X1V=1	in_region([X],R)	74	1.0	0.0	4	0.0
S9	X2V=0 ^ X2F=n/a ^ X1V=2	simple(action3(A,B))	707	1.0	0.0	74	0.0
S10	X2V=0 ^ X2F=n/a ^ X1V=0	simple(action5)	970	1.0	0.0	110	0.0
S11	X2V=0 ^ X2F=multiple ^ X1S=unbound	number(X,multiple)	1	9.0	0.0	153	0.54
S12	X2V=0 ^ X2F=type	holds(.,type([eye3],eye))	7	20.57	17.82	0	0.0
S13	X2V=0 ^ X2F=type	holds(.,size([eye3],[1,2]))	40	0.33	0.47	4	0.44
S14	X2V=0 ^ X2F=visible ^ X1F=result ^ X1V=1	holds(result(A,now),visible([eye3]))	4	2.25	1.30	1	1.25
S15	X2V=0 ^ X2F=visible ^ X1F=result ^ X1V=0	holds(result(action5,now),visible([eye3]))	896	0.09	0.34	119	0.44
S16	X2V=0 ^ X2F=visible ^ X1F=now	holds(now,visible([eye3]))	492	0.55	0.50	51	0.50
S17	X2V=0 ^ X2F=shorter_than	region(.,shorter_than,.)	154	1.0	0.0	19	0.0
S18	X2V=0 ^ X2F=list ^ X1F=action ^ X1V=2	target(action3(A,B),[eye3])	2025	0.01	0.10	249	0.09
S19	X2V=0 ^ X2F=list ^ X1F=action ^ X1V=0	target(action5,[eye3])	1408	0.03	0.18	150	0.18
S20	X2V=0 ^ X2F=list ^ X1F=n/a	target(A,[eye3])	23	3.0	0.0	4	0.0
S21	X2V=0 ^ X2F=list ^ X1F=list	in_region([1,2],[[0,1],[3,3]])	93	1.0	0.0	13	0.0
S22	X2V=2	holds(.,size(X,S))	1	991.0	0.0	0	0.0
	Other (no applicable rule)					0	0.0
	Overall		8071	1.12	13.02	919	2.20

Time required to find all solutions (TIME)							
ID	Rule	Example	Training Data			Test Data	
			N	Mean (ms)	STD	N	RMSE
T1	X2F=fits_plan ^ X2V=1	holds(.,fits_plan(A))	76	3.16	4.65	9	4.72
T2	X2F=fits_plan ^ X2V=0	holds(.,fits_plan(action5))	534	1.95	3.96	65	3.36
T3	X2F=1 ^ X1S=unbound	number(X,1)	3	0.0	0.0	0	0.0
T4	X2F=1 ^ X1S=bound	number([eye3],1)	46	0.22	1.46	6	0.22
T5	X2F=n/a ^ X1V=3	simple(action2(A,B,C))	43	0.47	2.11	3	0.47
T6	X2F=n/a ^ X1V=1	in_region([X],R)	73	0.68	2.53	5	4.21
T7	X2F=n/a ^ X1V=2 ^ predicate=target	target(action3(A,B),X)	23	0.87	2.82	2	0.87
T8	X2F=n/a ^ X1V=2 ^ predicate=simple	simple(action3(A,B))	681	0.18	1.32	75	1.15
T9	X2F=n/a ^ X1V=0 ^ X1S=unbound	simple(A)	14	0.0	0.0	1	0.0
T10	X2F=n/a ^ X1V=0 ^ X1S=bound	simple(action5)	944	0.32	1.75	121	1.28
T11	X2F=shape	holds(.,shape(X,rectangle))	1	890.0	0.0	17	0.38
T12	X2F=type ^ X2V=1	holds(.,type(X,eye))	4	912.5	12.99	3	29.55
T13	X2F=size ^ X2V=1	holds(.,size(X,[1,2]))	232	1.29	3.36	29	4.12
T14	X2F=size ^ X2V=2	holds(.,size(X,S))	1	2190.0	0.0	3	1.71
T15	X2F=visible ^ X1V=1 ^ X2V=1	holds(result(A,now), visible(X))	1	1120.0	0.0	1	0.0
T16	X2F=visible ^ X1V=1 ^ X2V=0	holds(result(A,now), visible([eye3]))	5	902.0	11.66	4	427.66
T17	X2F=visible ^ X1V=0 ^ X1F=result ^ X2V=0	holds(result(action5,now), visible([eye3]))	921	31.94	76.16	94	99.99
T18	X2F=visible ^ X1V=0 ^ X1F=now ^ X2V=1	holds(now, visible(X))	20	12.5	15.45	2	10.31
T19	X2F=visible ^ X1V=0 ^ X1F=now ^ X2V=0	holds(now, visible([eye3]))	495	1.13	3.17	48	2.47
T20	X2F=shorter_than ^ X3S=unbound ^ X1S=unbound	region(R,shorter_than,S)	102	0.59	2.35	15	3.48
T21	X2F=shorter_than ^ X3S=unbound ^ X1S=bound	region([1,2],[3,4],shorter_than,S)	6	0.0	0.0	2	6.30
T22	X2F=shorter_than ^ X3S=bound ^ X1S=bound	region([1,2],[3,4],shorter_than,[3,4])	4	0.0	0.0	1	0.0
T23	X2F=list ^ X1F=action ^ X1V=2	target(action3(A,B), [eye3])	2055	0.25	1.56	219	1.34
T24	X2F=list ^ X1F=action ^ X1V=0	target(action5, [eye3])	1407	0.18	1.35	151	1.96
T25	X2F=list ^ X1F=n/a	target(A, [eye3])	25	0.8	2.71	2	0.8
T26	X2F=list ^ X1F=list	in_region([1,2],[[0,1],[3,3]])	94	0.43	2.02	12	2.79
	Other (no applicable rule)					0	0.0
	Overall		8100	6.40	54.87	890	43.42

Figure 1. Regression trees for predicting number of constraint solutions (SOL) and time required to find them (TIME). The regression trees are illustrated as sets of rules for easier comprehension. Each rule in the figure corresponds to a leaf node in the regression tree learned by the CART algorithm. In the Example column, an underscore (.) for an argument indicates that that argument's value is not constrained by the relevant rule. An uppercase letter indicates an unbound variable in a position consistent with the attribute tests in the relevant rule. A lowercase term indicates a typical value for a bound argument consistent with the attribute tests in the relevant rule. Statistics are given for the training data (90%) and test data (10%) used to learn the regression trees indicated here. (Separate partitions were used to learn the two trees.) For the training data, the number of training examples (N) and the mean (Mean) and standard deviation (STD) in output values are indicated. For the test data, the number of test examples (N) and the root mean squared error (RMSE) of the regression tree on the test examples is indicated.