From: AAAI-90 Proceedings. Copyright ©1990, AAAI (www.aaai.org). All rights reserved.

# The Complexity of Constraint Satisfaction in Prolog

**Bernard A. Nadel**

Department of Computer Science
Wayne State University
Detroit, Michigan, 48202
ban@cs.wayne.edu

## Abstract

We obtain here the complexity of solving a type of Prolog problem which Genesereth and Nilsson have called *sequential constraint satisfaction.* Such problems are of direct relevance to relational database retrieval as well as providing a tractable first step in analyzing Prolog problem-solving in the general case. The present paper provides the first analytic expressions for the expected complexity of solving sequential constraint satisfaction problems. These expressions provide a basis for the formal derivation of heuristics for such problems, analogous to the theory-based heuristics obtained by the author for traditional constraint satisfaction problem-solving. A first application has been in providing a formal basis for Warren's heuristic for optimally ordering the goals in a conjunctive query. Due to the incorporation of "constraint looseness" into the analysis, the expected complexity obtained here has the useful property that it is usually quite accurate even for individual problem instances, rather than only for the assumed underlying problem class as a whole. Heuristics based on these results can be expected to be equally instance-specific. Preliminary results for Warren's heuristic have shown this to be the case.

## 1. Introduction

Two areas of major interest in Artificial Intelligence are *constraint satisfaction* problem solving and the logic programming language *Prolog.* Recently there has been considerable interest in the relationship between these two areas [4] [14] [16] [17] [18]. This paper grew out of a study of this relationship. It treats one of the simplest types of Prolog problems, which Genesereth and Nilsson [2] call *sequential constraint satisfaction problems.* These involve a database consisting of no rules, but only of facts all of which are ground, and a query which is a conjunction of positive literals containing variables. Strategies for ordering the conjunctive queries have been studied in [15] and [19].

The present paper provides the first analytic expressions for the expected complexity of solving such problems. The analysis is based on a division of the problem class into what we call *small classes*, defined in section 2. These classes, and the assumed probability model defined in section 3, are analogs for Prolog of those used in the author's earlier analyses of the complexity of traditional constraint satisfaction problem-solving [6], [8], [12]. As opposed to the *big classes* which were also used in prior work (but which are not defined here), most instances of a small classes have remarkably similar complexity values.

This *homogeneity* property is the result of taking into account "constraint tightness" (in terms of $S_j$ or $R_j$ below) in defining small classes. As a result of small class homogeneity, the expected case complexity for a given small class is likely to be a good estimate of the "exact case" complexity of most individual instances in the small class. This makes the expected case results much more practical by allowing them to be used for complexity prediction on an instance-by-instance basis, independent of the applicable probability distribution, if any, rather than only for a small class as a whole, under the assumed distribution.

The expected complexity for solving sequential constraint satisfaction problems is obtained in section 4, in terms of the expected number of nodes in the Prolog search tree for finding all solutions of a problem in a small class. Extensions of the results are discussed briefly in section 5. A simple running example is used throughout.

The complexity expressions obtained here provide a formal basis for the derivation of heuristics for solving this kind of Prolog problem, analogous to the theory-

based derivation of heuristics (for choosing a good algorithm, search ordering and even problem representation) which has been possible for regular constraint satisfaction problem-solving [3] [6] [7] [8] [9] [12] [13]. A first example of this for our Prolog problems has been in providing a formal basis for Warren's intuitively motivated heuristic for optimally ordering the goals in a conjunctive query [19]. The instance-specificity of the complexity results here is expected to carry over to the theory-based heuristics derived from them. Preliminary results for Warren's heuristic show this to be the case. By generalizing the problem class under consideration (in particular, allowing databases with rules and variables) it should be possible to obtain similar instance-specific, theory-based guidance for Prolog problem solving in general.

## 2. Sequential Constraint Satisfaction Problems and Small Classes

To analyze the complexity of solving sequential constraint satisfaction problems it is convenient to divide[1] the space of all possible databases for such problems into what we class small classes. The expected complexity derived will be over the databases of a small class, each database arising with equal probability as described in the next section.

Associated with a small class is an underlying set $Z = \{z_1, z_2, \ldots, z_n\}$ of $n$ variables, each having a finite domain $d_{z_i} = \{1, 2, \ldots, m_{z_i}\}$ of $m_{z_i}$ candidate values. We denote the domain values as integers here for convenience, but the results apply for values that are arbitrary Prolog ground terms. Variables $z_i$ will sometimes be called *domain variables* or *formal variables* to distinguish them from *logical variables* which appear in the queries to be asked with respect to a database.

There is a set $P = \{p_1, p_2, \ldots, p_K\}$ of $K$ predicate symbols, each having an associated *fact template*

$$p_j(z_{i_1}, z_{i_2}, \ldots, z_{i_{A_j}}) \tag{1}$$

which specifies its *arguments variables* $z_{i_k} \in Z$ and their corresponding argument positions. The order of arguments in a template is usually, but not necessarily, in increasing order of the argument variable indices. A fact template is analogous to a *relation schema* in relational database theory [5]. It is *not* the same thi as a literal used in a query of the Prolog databa although they are related as discussed below. $Z_j$ $\{z_{i_1}, z_{i_2}, \ldots, z_{i_{A_j}}\}$ and $A_j = |Z_j|$ are respectively t *argument set* and *arity* of $p_j$. We may assume witho loss of generality that the set $Z$ of all variables is giv by $Z = \bigcup_{j=1}^{K} Z_j$.

Each database of the small class consists only ground facts for each of the predicate symbols $p_j$. T fact templates serve to specify the form and range these facts. Specifically, facts for $p_j$ must have the fo

$$p_j(\bar{z}_{i_1}, \bar{z}_{i_2}, \ldots, \bar{z}_{i_{A_j}})$$

where $\bar{z}_{i_k}$ is a value chosen only from the domain the argument variable $z_{i_k}$ occurring for that positi in the fact template. Each fact $p_j(\bar{Z}_j)$ thus correspon to an $A_j$-tuple $\bar{Z}_j$ from the cartesian product $D_j$ of t domains of the arguments of $p_j$. Cartesian product and its size $M_j$ are given respectively by $D_j = \bigtimes_{z_i \in Z_j}$ and $M_j = \prod_{z_i \in Z_j} m_{z_i}$.

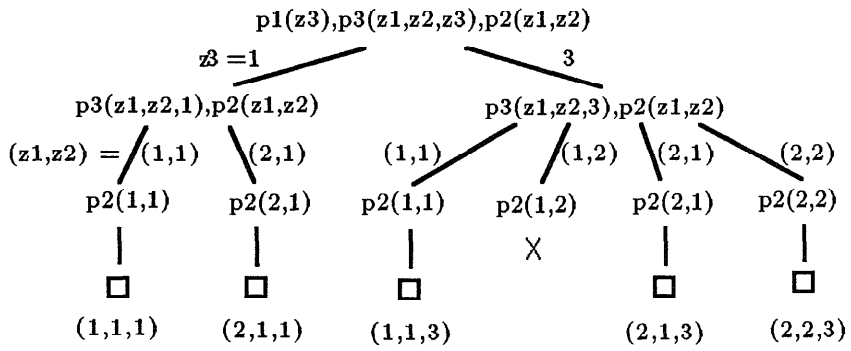
For each $p_j$ there is a parameter $S_j$, called the *satisfiability* of $p_j$, which specifies how many facts for are in the database. The quantity $R_j = S_j/M_j$ call the *satisfiability ratio* for $p_j$. Of course $S_j$ and respectively satisfy $0 \leq S_j \leq M_j$ and $0 \leq R_j \leq$ $R_j$ is the fraction of possible tuples (formable fr the domains of the corresponding argument variabl which actually correspond to facts in the database $p_j$. Considering the goals of a conjunctive query constraints to be simultaneously satisfied, $R_j$ is a m sure of the looseness of the constraint corresponding the $p_j$ goal.

The required $S_j$ facts for $p_j$ can correspond to a size-$S_j$ subset of $D_j$ and hence there are $\binom{M_j}{S_j}$ w to choose the facts for $p_j$. Since such a choice of facts is required for each $p_j$, $1 \leq j \leq K$, the numl of sequential constraint satisfaction databases scsd a small class is

$$|\mathrm{SCSD}(n, \mathbf{m}, K, \mathbf{Z}, \mathbf{S})| = \prod_{j=1}^{K} \binom{M_j}{S_j}.$$

Here $\mathrm{SCSD}(n, \mathbf{m}, K, \mathbf{Z}, \mathbf{S})$ denotes the generic sm class characterized by $n$ problem variables, set $\mathbf{m}$ $\{m_{z_1}, m_{z_2}, \ldots, m_{z_n}\}$ of domain sizes, $K$ predicate sy bols, set $\mathbf{Z} = \{Z_1, Z_2, \ldots, Z_K\}$ of predicate argum sets and set $\mathbf{S} = \{S_1, S_2, \ldots, S_K\}$ of predicate satis bilities.

[1] This division is not a partition however, since a given database may be considered as a member of an infinite number of small classes, as implied in section 5 and discussed more fully in [11].




| $k$ | $N_k$ | $\bar{N}_k$ | | |
|---|---|---|---|---|
| 0 | 1 | 1 | = | 1 |
| 1 | 2 | $1 \cdot (3 \cdot \frac{2}{3})$ | = | 2 |
| 2 | 6 | $2 \cdot (4 \cdot \frac{10}{12})$ | = | $6\frac{2}{3}$ |
| 3 | 5 | $6\frac{2}{3} \cdot (1 \cdot \frac{3}{4})$ | = | 5 |
| | 14 | | | $14\frac{2}{3}$ |

**Fig. 1:** The Prolog search tree for the scsp of solving query (2) with respect to our example database.

For example, consider the set $Z = \{z_1, z_2, z_3\}$ of $n = 3$ underlying domain variables, having domains $d_{z_1} = d_{z_2} = \{1,2\}$ and $d_{z_3} = \{1,2,3\}$ of respective sizes $m_{z_1} = m_{z_2} = 2$ and $m_{z_3} = 3$. Let's say the database consists of facts for $K = 3$ predicate symbols $p_1$, $p_2$, $p_3$, with respective fact templates $p_1(z_3)$, $p_2(z_1, z_2)$, $p_3(z_1, z_2, z_3)$ and satisfiabilities $S_1 = 2$, $S_2 = 3$, $S_3 = 10$. The cartesian products for the three predicates therefore have respective sizes $M_1 = m_{z_3} = 3$, $M_2 = m_{z_1} \times m_{z_2} = 4$ and $M_3 = m_{z_1} \times m_{z_2} \times m_{z_3} = 12$. Thus the number of databases in the small class is $\binom{M_1}{S_1} \times \binom{M_2}{S_2} \times \binom{M_3}{S_3} = \binom{3}{2} \times \binom{4}{3} \times \binom{12}{10} = 792$. The following is one of these 792 possible Prolog databases:

$p_1(1).\ p_1(3).$ $\quad p_2(1,1).\ p_2(2,1).\ p_2(2,2).$
$p_3(1,1,1).\ p_3(1,1,2).\ p_3(1,1,3).\ p_3(1,2,2).\ p_3(1,2,3).$
$p_3(2,1,1).\ p_3(2,1,2).\ p_3(2,1,3).\ p_3(2,2,2).\ p_3(2,2,3).$

As required by the corresponding fact templates, the values for the single argument of $p_1$ come from the domain of $z_3$, the values for the first and second arguments of $p_2$ are respectively from the domains of $z_1$ and $z_2$, and the first, second and third argument values for $p_3$ are respectively from the domains of $z_1$, $z_2$ and $z_3$.

A sequential constraint satisfaction problem *scsp* is a sequential constraint satisfaction database *scsd* plus an associated conjunctive query $Q$ such as

$$?\text{-}\ p_1(z_3), p_3(z_1, z_2, z_3), p_2(z_1, z_2). \qquad (2)$$

The $z_i$ here are logical variables, not necessarily related to the domain variables of the same name used to define the small class. Figure 1 shows the Prolog search tree for solving this query with respect to the above database. Success nodes, indicated by $\square$, are labelled by the corresponding 3-tuple solution for the logical variables in the order $(z_1, z_2, z_3)$. In the following sections we derive the expected case complexity of answering such queries with respect to the databases of a small class. The analysis is for the case, as in figure 1, where *all* solutions of a query are sought.

## 3. Probability Model

The probability model we assume for the databases of a small class is very straight-forward: each selection of $S_j$ tuples from the cartesian product $D_j$ of the $j$-th constraint is equally likely, and such a sclection is made independently for each predicate $p_j$, $1 \leq j \leq K$. As such, each database scsd in a small class SCSD is equally likely, with probability $P(\text{scsd}) = |\text{SCSD}|^{-1} = \prod_{j=1}^{K} \binom{M_j}{S_j}^{-1}$.

## 4. Expected Complexity

For the generic small class SCSD$(n, \mathbf{m}, K, \mathbf{Z}, \mathbf{S})$ described above, our analysis considers only queries which are a conjunction, in any order, of $K$ positive literals, one for each small class predicate symbol $p_j$. These literals contain logical variables. In practice, logical variables are not necessarily named $z_i$ as are the domain variables in terms of which a small class is defined. Nor is there necessarily a one to one correspondence between logical variables in the set of query literals and domain variables in the set of fact templates. For example, given our previous example small class, such a correspondence is lacking for query

$$?\text{-}\ p_1(z_1), p_2(z_1, z_3), p_3(z_3, z_1, z_2) \qquad (3)$$

even though this is a perfectly legal query. However, for the following analysis we assume queries for which such a one to one correspondence does exist. For simplicity we may then assume that logical variables have the same names as their corresponding domain variables. Therefore, if in the small class definition, a predicate has associated fact template as in (1), then we assume an "isomorphic" corresponding literal

$$p_j(Z_j) = p_j(z_{i_1}, z_{i_2}, \ldots, z_{i_{A_j}}), \tag{4}$$

in the query, the $z_i$ here of course being logical variables. As in (4), we may therefore also use $Z_j$ to denote the set of logical variables of the $p_j$ query literal, as well as for the set of domain variables in the $p_j$ fact template.

To allow for an arbitrary permutation $\pi$ of conjunctions (literals) in the query, we write $p^j$ and $Z^j$ respectively for the predicate symbol and the argument set of the $j$-th literal in the query. The queries we consider are thus of the form

$$?- p^1(Z^1), p^2(Z^2), \ldots, p^K(Z^K) \tag{5}$$

where $p^j = p_{\pi(j)}$ and $Z^j = Z_{\pi(j)}$ (and similarly for all predicate-related quantities: $D^j = D_{\pi(j)}$, $M^j = M_{\pi(j)}$, $S^j = S_{\pi(j)}$, $R^j = R_{\pi(j)}$). Query (2) used in figure 1 is of the form given by (4) and (5), assumed in our analysis. It corresponds to permutation $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{smallmatrix}\right)$ and we have $p^1 = p_{\pi(1)} = p_1$, $p^2 = p_{\pi(2)} = p_3$ and $p^3 = p_{\pi(3)} = p_2$, and analogously $Z^1 = Z_1$, $Z^2 = Z_3$ and $Z^3 = Z_2$. Another query of the form covered by our analysis is

$$?- p_2(z_1, z_2), p_1(z_3), p_3(z_1, z_2, z_3).$$

corresponding to permutation $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{smallmatrix}\right)$.

As our measure of complexity for solving sequential constraint satisfaction problems in Prolog we use the number of nodes in the Prolog search tree for finding all solutions. We include the root node and the success nodes $\square$. Variations on this measure may be appropriate for different implementations of Prolog, but in general these variations should be closely related to the fundamental measure used here and be derivable by a similar analysis.

Counting the root node as being at level $k = 0$, the search tree for solving a scsp has levels $k = 0, 1, \ldots, K$, as for example in figure 1. Since the number of nodes $N$ in a search tree can of course be expressed as the sum of the number of nodes $N_k$ at each level $k$ of the tree we have $N = \sum_{k=0}^{K} N_k$. Taking the expectation of both sides we have that the expected number of nodes in a Prolog search tree for instances of a small class of $K$ predicates is the sum of the expected number $\bar{N}_k$ of nodes at each level,

$$\bar{N} = \sum_{k=0}^{K} \bar{N}_k. \tag{6}$$

We now obtain a recursive, then non-recursive, expression for $\bar{N}_k$. The nodes at level $k$ arise as children of those at level $k-1$. These children of a node are generated by unifying in all ways the leftmost literal at a level $k-1$ node with the corresponding facts in the database. The leftmost literal at a level $k-1$ node is the $p^k$ literal (that corresponding to predicate symbol $p^k$) and each node at level $k-1$ potentially has a child for each combination of values that can be assigned to the uninstantiated variables of that literal. The set of *instantiated* variables is the same for the leftmost literal of each level $k-1$ node, and is the union $I^{(k)} = \bigcup_{j=1}^{k-1} Z^j$ of the argument sets of the initial-query literals $p^1(Z^1)$ to $p^{k-1}(Z^{k-1})$, since these literals have all been solved at preceding levels. The set of *uninstantiated* variables of the $p^k$ literal at a level $k-1$ node is thus $U^{(k)} = Z^k - I^{(k)}$, the set difference between the argument set for the literal and the instantiated variables so far.

By definition of the parent small class, there are $m_{z_i}$ à priori possible values for variable $z_i$. Each uninstantiated variable $z_i \in U^{(k)}$ of the $p^k$ literal at a level $k-1$ node has such an associated domain, and hence

$$m^{(k)} = \prod_{z_i \in U^{(k)}} m_{z_i} \tag{7}$$

children are à priori possible for a level $k-1$ node by unifying each uninstantiated variable in all ways against facts in the database. However, also by definition of the parent small class, only some fraction $R^k = S^k/M^k$ of the possible $M^k$ tuples of the cartesian product $D^k$ correspond to database facts for $p^k$. Thus only this fraction of the maximum $M^k$ unifications are possible (the actual number of possible unifications being $S^k = R^k M^k$) for the initial-query literal $p^k(Z^k)$. Even though in general the $p^k$ literal at level $k-1$ is not $p^k(Z^k)$, but is $p^k(Z^k)$ with some argument variables $z_i \in Z^k$ already instantiated, the same fraction $R^k$ nevertheless governs how many unifications are possible on average for the $p^k$ literal at level $k-1$ against database facts, given an arbitrary assignment of values to its instantiated variables. But due to the

instantiation of some variables of the $p^k$ literal, this fraction is now out of $m^{(k)}$ possibilities rather than all $M^k$.

The average number of children (or average branching factor) of a level $k-1$ node is therefore $m^{(k)}R^k$. The average number of nodes at level $k$ is thus $\bar{N}_k = \bar{N}_{k-1}m^{(k)}R^k$, the average number of nodes at the previous level by their average branching factor. We have then the following recurrence for the expected number of nodes at level $k$ in the Prolog search tree

$$\bar{N}_k = \begin{cases} 1 & \text{if } k = 0 \\ \bar{N}_{k-1}m^{(k)}R^k & \text{if } 1 \leq k \leq K \end{cases} \quad (8)$$

which, by induction, has the closed-form solution

$$\bar{N}_k = \prod_{j=1}^{k} m^{(j)} R^j = \prod_{j=1}^{k} \Big[ \prod_{z_i \in U^{(j)}} m_{z_i} \Big] R^j. \quad (9)$$

A product over no terms (i.e. when $k = 0$) is considered to equal 1 (cf. $x^0 = 1$). Similarly in (7), if there are no uninstantiated argument variables for the $p^k$ literal at level $k-1$ then $m^{(k)} = 1$. By (6) and (9), the expected total number of nodes in a Prolog search tree is

$$\bar{N} = \sum_{k=0}^{K} \prod_{j=1}^{k} m^{(j)} R^j = \sum_{k=0}^{K} \prod_{j=1}^{k} \Big[ \prod_{z_i \in U^{(j)}} m_{z_i} \Big] R^j \quad (10)$$

Note that for $k = K$, expression (9) must be the expected number of solutions for an instance of a small class — a result which (unlike $\bar{N}_k$ for $k < K$) is of course independent of the ordering of the $K$ conjuncts in the initial query. Since by assumption, $Z$ is the set of all variables appearing in the $K$ literals of the initial query, and the set of uninstantiated variables at the leftmost literal over all levels must be $Z$, we have from (9) when $k = K$, that the expected number of solutions is

$$\bar{S} = \Big[ \prod_{z_i \in Z} m_{z_i} \Big] \Big[ \prod_{j=1}^{K} R_j \Big].$$

Figure 1 includes a table at the right showing the number of nodes $N_k$ at each level $k$ for the particular problem instance being solved and also the corresponding expected number of nodes $\bar{N}_k$ by (8) at level $k$ for the small class to which the instance belongs. For example, in calculating the expected number of nodes $\bar{N}_2$ at level $k = 2$, there are 2 uninstantiated variables, $z_1$ and $z_2$, in the leftmost $p^2 = p_3$ literal at the preceding level $k = 1$. We thus have $m^2 = m_{z_1} \times m_{z_2} = 4$, and from the initial specification of the small class we have $R^2 = R_3 = S_3/M_3 = 10/12$. Thus $(4 \cdot \frac{10}{12})$ is the second factor in the expression for $\bar{N}_2$ in the table of figure 1. By (8), the first factor is $\bar{N}_1$ from the preceding line of the table.

## 5. Applications and Extensions

Due to the preliminary nature of the work and also because of space restrictions, we only sketch here the ways in which the above complexity results may be applied and extended. This section is based on [11] where these issues have been addressed in more detail.

The above complexity expressions are for finding all solutions to a conjunctive query against databases of an arbitrary small class, where the databases are all equally likely. As such, our complexities are exact *expected case* results. However, we have found them also to be good approximations to the *exact case* complexity of solving most individual instances of a given small class.

Experiments have shown that most instances of a small class have a similar complexity of solution, and so the expected case complexity for a small class is close to that of most subsumed instances. Specifically, we have found that about 85% of the instances in a small class have (exact-case) complexity of solution within 15% of the expected complexity for their small class value. It is in this sense that we say that small classes are *homogeneous*. Homogeneity is an unusual and very useful property of problem subclasses. It was first observed for the essentially same kind of small classes (called *c-classes* in [12]) used in our analyses of traditional (non-Prolog) methods for solving constraint satisfaction problems [6] [8] [12].

The key to obtaining homogeneity for constraint satisfaction problems is the use of the $R_j$ "constraint looseness" parameters in defining small classes. Problem instances whose corresponding "constraints" have equal looseness are usually sufficiently alike, given equality of the other parameters used in defining a small class, that their complexities of solution are also close. Small class homogeneity allows our expected case complexity to be used — as in figure 1 — as a good estimate for individual instances of the corresponding small class. As such, the results can be applied irrespective of the actual probability distribution or grouping of instances that occurs in practice. The notion of homogeneity and its importance is studied more fully in [10].

However our analysis assumed a query of quite re-

stricted form; in particular we required literals to be isomorphic to the fact templates defining the small class (see (4)). More general kinds of queries are certainly possible, such as that in (3). Our results do not give the *expected* complexity for solving such queries.

However, in approximating the *exact* case complexity for a specific problem instance our results may be applied even for these different kinds of queries. The reason is simply because when only a single instance is of interest, no particular parent small class has been specified. The latter may therefore be chosen as convenient for the purposes of the computation. In particular, we are free to choose the parent small class so that the database query of interest satisfies the required assumptions of the analysis. The procedure is quite straight-forward, as given in [11]. Having chosen an appropriate parent small class, the corresponding parameter values are inserted into (10). Due to homogeneity, the resulting expected complexity for the small class is likely to be a good approximation for the original instance even though the small class is only a "nominal" parent of the instance. In this way small class homogeneity allows our results to be used to approximate well the *exact* case complexity of most scsp instances, even for conjunctive queries not of the type assumed for the *expected* case analysis per se.

The next logical step is to consider ways in which our results may guide problem solving so as to *minimize* complexity, rather than just *predicting* the complexity. Warren in [19] has proposed a heuristic for ordering the conjuncts in a query: rank the literals according to increasing cost, where cost of a literal (he calls them *goal predicates*) is defined as "the total size (i.e. number of tuples) of the relation corresponding to the goal predicate, divided by the product of the sizes of the domains of each instantiated argument position". It is not obvious why his cost measure, and its use in this way, is appropriate. However it becomes clear when the relation to our analysis is established. In our notation Warren's cost for the $k$-th literal in a query is

$$\begin{aligned} S^k / \prod_{z_i \in I^{(k)}} m_{z_i} &= S^k / \prod_{z_i \in Z^k - U^{(k)}} m_{z_i} \\ &= S^k \Big[ \prod_{z_i \in U^{(k)}} m_{z_i} \Big] / M^k \\ &= \Big[ \prod_{z_i \in U^{(k)}} m_{z_i} \Big] R^k \\ &= m^{(k)} R^k \end{aligned}$$

This is just the branching factor from (8). In other words, Warren's heuristic can be interpreted as

1. implicitly assuming that the instance of interest is a member of its natural parent small class, uniformly distributed according to our probability model above
2. choosing the next literal using a "greedy" approach of minimizing the expected (over the parent small class) branching factor, and hence the expected number (over the parent small class) of nodes at the next level
3. implicitly taking advantage of small class homogeneity in using the result obtained with respect to the parent small class as appropriate to the initial instance itself.

Our experiments [1] have shown Warren's heuristic to usually lead to the optimal, or a near optimal, conjunct ordering in terms of minimizing the expected total number of nodes $\bar{N}$ for the assumed parent small class of a given instance. Moreover, as a result of small class homogeneity, the actual total number of nodes $N$ for a given instance is usually close to $\bar{N}$ for its parent small class, so that the conjunct ordering which minimizes $\bar{N}$ also tends to minimize $N$ for the instance.

However, Warren's heuristic does not *always* succeed in minimizing $\bar{N}$ of a small classes, let alone in minimizing $N$ of an individual member instance. More accurate heuristics are however implicit in our results above. One just needs to use (10) more completely, rather than in the incomplete greedy manner in which it was implicitly used by Warren. We are currently investigating such improved theory-based heuristic guidance in solving sequential constraint satisfaction problems [1]. Moreover, by generalizing the problem class under consideration (in particular, to allow databases with rules and variables) it should be similarly possible to obtain a formal basis for instance-specific, theory-based optimization of Prolog problem solving in general.

## References


[1] Chugh K., *Theory-based heuristics for constraint satisfaction in Prolog*, Computer Science Dept., Wayne State University, Detroit, Michigan, M. Sc. thesis, 1990, to appear.

[2] Genesereth M. R., and Nilsson N. J., "Logical Foundations of Artificial Intelligence", *Morgan Kaufmann*, Los Altos, California., 1987.

[3] Haralick R. M. and Elliot G. L., "Increasing tree search efficiency for constraint satisfaction problems", *Artificial Intelligence*, vol. 14, 1980, pp. 263-313.

[4] Jaffar J. and Lassez J.-L., "Constraint logic programming", *Proc. 14-th ACM Conference on Principles of Programming Languages*, Munich, West Germany, January, 1987.

[5] Maier D., *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[6] Nadel B. A., *The Consistent Labeling Problem and its Algorithms: Towards Exact-Case Complexities and Theory-Based Heuristics*, Computer Science Dept., Rutgers University, New Brunswick, N. J., May 1986, Ph. D. dissertation.

[7] Nadel B. A., "Representation selection for constraint satisfaction: a case study using $n$-queens", *IEEE Expert*, vol. 5, #3, June 1990.

[8] Nadel B. A., "The complexity of Backtracking and Forward Checking: search-order and instance specific results", submitted for publication. Also in technical report CSC-88-002, Dept. Computer Science, Wayne State University, Detroit, Michigan.

[9] Nadel B. A., "Constraint satisfaction algorithms", *Computational Intelligence*, vol. 5, no. 4, November 1989, pp. 188-224. A preliminary version appeared as "Tree search and arc consistency in constraint satisfaction algorithms", in *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, 1988.

[10] Nadel B. A., "Precision complexity analysis: a case study using insertion sort", submitted for publication. Available as technical report CSC-88-008, 1988, Dept. Computer Science, Wayne State University, Detroit, Michigan.

[11] Nadel B. A., "The complexity of constraint satisfaction in Prolog", technical report CSC-89-004, 1989, Dept. Computer Science, Wayne State University, Detroit, Michigan.

[12] Nudel B. A., "Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics", *Artificial Intelligence (Special Issue on Search and Heuristics)*, vol. 21, nos. 1 and 2, March 1983, pp. 135-178. Also in book: *Search and Heuristics*, Ed. J. Pearl, North-Holland, Amsterdam, 1983.

[13] Nudel B. A., "Solving the general consistent labeling (or constraint satisfaction) problem: Two algorithms and their expected complexities", *Proc. Nat. Conf. on Artificial Intelligence (AAAI)*, Washington D.C., August 1983, pp. 292-296.

[14] Rossi F., "Constraint satisfaction problems in logic programming" SIGART Newsletter of the ACM, October 1988, pp. 24-28.

[15] Smith D. E. and Genesereth M. R., "Ordering conjunctive queries", *Artificial Intelligence*, vol. 26, no. 3, 171-215, 1985.

[16] Van Hentenryck P. and Dincbas M., "Domains in logic programming", *Proc. Fifth Nat. Conf. on Artificial Intelligence (AAAI)*, Philadelphia, Pennsylvania, August 1986.

[17] Van Hentenryck P., *Consistency Techniques in Logic Programming*, Universitaires Notre-Dame de la Paix, Namur, Belgium, July 1987, Ph. D. dissertation.

[18] Van Hentenryck P., "A theoretical framework for consistency techniques in logic programming", *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, August 1987, pp. 2-8.

[19] Warren D., "Efficient processing of interactive relational database queries expressed in logic", *Proc. Seventh Conf. on Very Large Data Bases*, Cannes, France, 272-282, August 1981.