Arbeitspapiere des SFB 340
Bericht Nr. 140, March 1999

OPTIMISING DON'T-CARE NON-DETERMINISM WITH STATISTICAL INFORMATION

GERALD PENN

Abstract. A general statistical method for improving the performance of compiled code is presented, along with its preliminary application and evaluation to three sources of non-determinism that arise with natural language processing in ALE (Carpenter and Penn 1996), a logic programming language based on the logic of typed feature structures (Carpenter 1992).

Contents

1. Introduction: Don't-Care Non-Determinism

Suppose we are required to perform a task, G, consisting of n sub-tasks, $F_1, \ldots, F_n$, such that G succeeds iff all of $F_1, \ldots, F_n$ succeed, with no additional requirements on the order in which $F_1, \ldots, F_n$ are attempted. This is an instance of *don't-care non-determinism (DCND)* — we don't care in which order we accomplish the subtasks as long as all of them

ISSN 0947-6954/99

succeed. Many tasks in both natural language processing and compiler design are instances of DCND.

This paper describes a preliminary investigation into using statistical information to improve the efficiency of performing DCND tasks, with particular attention to DCND tasks that can found in natural language processing using the Attribute Logic Engine (ALE) (Carpenter and Penn 1996), a logic programming language based on typed feature structures.

If $F_1, \ldots, F_n$ are independent, and will all succeed (hence G succeeds), then the order in which $F_1, \ldots, F_n$ are performed does not, in general, effect the speed with which G is completed. If, on the other hand, we have prior knowledge that G will fail or is likely to fail, then we can deduce that this will happen because at least one of $F_1, \ldots, F_n$ will fail or is likely to fail. If we can choose which of $F_1, \ldots, F_n$ is most likely to fail, we can try that one first, and thereby avoid wasting time on the other subtasks before reporting that G has failed.

2. Some Examples of DCND and its Resolution

2.1. Context-Free Parsing. Of course, statistical methods have enjoyed a very wide use in natural language processing, but as a rule, they are not being applied to instances of DCND. Instead, they are being used to estimate explicit parameters in a language model that can be used to find a "best" solution first, or a first solution as quickly as possible. What we are interested in here is an application of statistically obtained preferences that works, in principle, with a program or grammar that is still entirely symbolic, although it is certainly compatible with these other approaches.

One of the few previous attempts to address DCND with statistical methods was (Gibson 1992), which observed that matching multiple categories on the right-hand side of a context-free rule was an instance of DCND in bottom-up parsing that could be optimised statistically. For example, given the rule:

$$S \longrightarrow NP \quad VP$$

both an NP and a VP must be found in order to hypothesis an S with this rule. Whether we choose to find the NP or the VP first does not matter, logically, since we need both. If the likelihood of finding an available NP significantly exceeds the likelihood of finding an available VP, however, it makes more sense to look for a VP first. If none is available in the position needed, then we do not need to search for an NP. (Gibson 1992) found that there are indeed significant enough differences in such likelihoods for natural languages and that an improvement in parsing times of up to a factor of 2 can be gained using these statistically determined likelihoods as preferences for resolving this instance of DCND.

2.2. **Pattern Matching in ML.** In the programming language, ML, a function can be defined by a sequence of pattern-expression pairs. When a function is invoked, it is handled by the first expression (in the order declared) whose pattern matches its argument. If no pattern matches the argument, an exception is raised. For example,[1]. the function `diff` on two lists can be defined by the following pairs:

```
fun diff (nil, y) = y |
    diff (x, nil) = x |
    diff (cons(hdx,tlx), cons(hdy,tly)) = diff (tlx,tly);
```

For an argument to match a pattern, their data constructors must be the same, and the subarguments must match the respective subpatterns. In the above example, the argument must be a pair (given by `(,)`) in every pattern. In the third pattern, both subarguments must also have `cons` constructors. The first and second patterns overlap on the pair `(nil,nil)`, in which case the first pattern applies. Provided that there is no overlap, we can check the subarguments in any order (even from different patterns) as long as they all match the respective subpatterns of one pattern in the end. This instance of DCND was resolved by (Baudinet and MacQueen 1985) by using heuristics based on prior knowledge to guide the choice of order. These include the *branching factor heuristic*, which says that we should test in the order of lowest branching factor first. If we have a type `color`, for example, with possible values, `red`, `green` and `blue`, and we need to match against the patterns of type `(bool * color)`:

```
(true,green)
(false,green)
```

then we should test the second subargument first, because only one of the possible `color` types (`green`) is attested, whereas both `true` and `false` are in the first subargument. If the value in the second subargument is not `green`, then we do not need to check the first.

The branching factor heuristic attempts to predict which subargument match is most likely to fail on the basis of static analysis of the type system. In the above example, it works, because matching the first argument can never fail. If, however, we had the patterns of type `(color*color)`:

[1]Most of the examples in this section are taken from (Baudinet and MacQueen 1985)

```
(red,green)
(blue,green)
```

instead, then it may not work. It could be, for example, that nearly all data objects that we encounter at run-time will have a value `green` for both their first and second subarguments. In that case, testing the second subargument will nearly always succeed. The best one to test first is the first, which will rule out most objects before the second needs to be tested. The branching factor heuristic relies on an assumption of a uniform empirical distribution of data objects across constructors or types in order to optimise this instance of DCND.

2.3. **Prolog Term/Predicate Unification.** Suppose we want to unify two Prolog terms. In order for unification to succeed, they must have the same principal functors and the same arities, and their respective subterms must unify. Every commercially available Prolog unifies subterms in order from left to right (Aït-Kaci 1991).

Prolog must also effectively unify predicates with the heads of clauses that define that predicate. As in the case of ML, the order in which clauses are declared matters when their heads overlap. Commercially available Prologs unify the arguments of predicates in order from left to right here too; but they do so along with the public service announcement that the first argument is treated specially. This special treatment is the well-known *first-argument indexing*, which allows prudent Prolog programmers to place the best discriminating argument for clause selection in first position to improve efficiency. This method not only assumes that programmers will always be able to do this, regardless of the size and nature of the knowledge domain, but that it is *always* the best first try, even when first arguments from two or more clauses overlap, and when terms that fall into that overlap are very frequently encountered. This requires the user to have an additional understanding not only of the kinds of first arguments, but of their frequencies in representative data. In the absence of this understanding, one can do no better than to make naive assumptions of uniform distributions of terms over first (or subsequent) argument patterns.

3. Applying Statistical DCND Resolution to ALE

ALE is a logic programming language with a built-in bottom-up chart parser and generator, based on the logic of typed feature structures. The programming language itself is almost identical to Prolog, although arguments are typed feature structures, rather than Prolog terms. The patterns that match these feature structures in predicate definitions come from a simple description language. ALE's parser and generator use an extended context-free rule formalism that also uses the description language to describe daughter and mother categories. The rule formalism also allows procedural attachments, and is essentially the equivalent of the definite clause rule formalism in Prolog.

ALE has a compiler which compiles its logic programming predicates into Prolog predicates, and its extended context-free rules into Prolog control code for parsing. As a result, grammars and their procedural attachments can be "compiled" into intermediate code for a parser or generator for that grammar that is then compiled further by a Prolog compiler. ALE itself is implemented in Prolog, so the former phase of compilation can be viewed simply as a pre-processing step to normal Prolog compilation.

3.1. **Unification.** ALE's type system already eliminates one source of non-determinism that many less restricted feature logics must deal with, namely the question at run-time of which features actually have values in a given feature structure. In ALE, a set of *appropriateness conditions* determines, for every type, exactly which features can and must have values for every feature structure of that type. ALE also has subtyping and multiple inheritance, so features can be added as the type of an object promotes, but for a fixed type, the set of features that occur is also fixed.

As a result, the Prolog term unification problem discussed above has an exact analogue in ALE. Given two typed feature structures of unifiable types (for expository reasons, I assume they are of the same type), in which order do we unify their appropriate features?

$$\begin{bmatrix} \text{INDEX} & \\ \text{PERSON} & \text{third} \\ \text{NUMBER} & \text{singular} \\ \text{GENDER} & \text{masc} \end{bmatrix} \sqcup \begin{bmatrix} \text{INDEX} & \\ \text{PERSON} & \text{third} \\ \text{NUMBER} & \text{plural} \\ \text{GENDER} & \text{masc} \end{bmatrix}$$

Currently ALE unifies them in alphabetical order by feature name — an arbitrary choice. It has been suggested (Schöter 1992) that it would be better to base the choice on an analysis of the values to which each feature is restricted (restriction of appropriate features to a particular type is also part of the appropriateness conditions). For example, the feature PERSON is restricted to the type *person*, and NUMBER is restricted to *number*:

```
person sub [first,second,third].
```

```
number sub [singular,plural].
```

We may think of preferring PERSON because its appropriate value has three subtypes, whereas NUMBER's only has two — there are more non-unifiable things at PERSON.

This heuristic relies on an assumption of a uniform empirical distribution of data objects across constructors or types in order to optimise this instance of DCND. If 95% of all *person*-typed feature structures are of type *third*, but *number*-typed feature structures are evenly split between *singular* and *plural*, then NUMBER is actually a better feature to choose. This problem is inherently an empirical one, and cannot be solved by static analysis alone. We could hope to gain from such an empirical study a better ordering for a fixed grammar/program, or homogeneous collection of grammars/programs, or perhaps even some better heuristic methods that could be applied to a large class of grammars/programs without the need for more data collection.

How do we collect the empirical data? Overhead is a serious concern. A naive method would be to compile a grammar/program once without empirical data, collect the data in a training phase for each appropriate feature over representative input, and then recompile using that data. That training phase, however, would need to perform unification in such a way that unification of every feature in a feature structure was attempted even if another had previously failed, which would make unification during training extremely slow. In addition, small changes to a grammar/program, particularly the declaration of its types and appropriate features, could conceivably obviate all previously collected statistics, which means that no such training could be performed until development was nearly complete.

A simpler method is to make an initial assumption of uniformity from which empirical data can gradually depart, so that training can occur during (mostly) normal unification. For each type, we can instrument compiled code to count the successes and failures for unification of each appropriate feature's value. After 10 failed unifications, a table of these counts might look as follows:

index:	S	F
PERSON	10	0
NUMBER	0	10
GENDER	0	0

No statistics have been collected yet on GENDER because all unifications so far have failed at NUMBER (assuming an initial ordering of PERSON, NUMBER, GENDER). For any two features, H_1 and H_2, such that H_1 precedes H_2 in the current ordering, if $(S_1 - S_2) - (F_1 - F_2)$ exceeds some threshold value, then H_1 and H_2 should be swapped in the ordering. For a suitably low threshold, the above table would result in a reordering to:

index:	S	F
NUMBER	0	10
GENDER	0	0
PERSON	10	0

The feature that is most likely to fail occurs first. The feature for which the uniform assumption of equal likelihood for success and failure comes next, and the "sure success" PERSON feature comes last. After a period of no reorderings exceeds another threshold value, the code can be de-instrumented to restore unification without overhead, but with the optimised ordering. Both thresholds must be empirically determined.

Note that this method only reorders the unification of sister features in a feature structure. In principle, it would be possible to reorder unification of substructures globally, or at least down to a further depth than 1, although this would require more counts to be kept.

3.2. **Indexing.** Another application of this resolution method is to the indexing problem. A parsing chart, for example, must be searched at run-time for feature structures that match a particular daughter description in a grammar rule. For semantic-head-driven generation, the lexicon needs to be indexed by feature structure (rather than by phonological string, as in parsing), and searched at run-time for a match to the semantic representation of the current semantic head. The use of an automaton-based indexing method for the latter task, along with suggestions for its improvement with a statistical method is given in (Penn and Popescu 1997). The former is a bit more difficult because the parsing chart itself is being built on-line.

Indexing can be viewed as a very special case of unification, in which one of the operands is at least partially known at compile-time. The above method is thus applicable, in principle. In general, not all types and features will need to be instrumented with counters, depending on how much of the information in a feature structure is referenced in the index itself.

The indexing problem is again, a necessarily empirical one. Other well-known non-statistical techniques exist for optimising the indexing problem, such as applying conjuncts of an index that are more deterministic (when indices can contain disjunctions, as they can in ALE), or pre-computing which mothers of rules could serve as daughters to other rules, which has the effect of pre-computing some indexing information. In the worst case, however, there can be a very large number of conjuncts that are all deterministic, and rules may be as simple as:

```
my_rule rule
(sem:Sem, subcat:[])
===>
cat> (sem:Sem, subcat:[Arg]),
cat> Arg.
```

This simple rule is not unlike what HPSG actually posits for some of its immediate dominance schemata, for example. Yet it says essentially nothing at compile-time about its second daughter category (with the description, `Arg`) that could be used to pre-compute a better index for it. An empirical study of what the variable `Arg` is typically bound to must be used to optimise indexing in cases such as this. The fact that ALE allows potentially recursive procedural attachments on its rules precludes using more ambitious packed forest representations of parsing charts in ALE, which has an effect on the indexing problem as well.

3.3. **Feature Unfolding.** The logic of typed feature structures essentially underdetermines the representation of many domains of knowledge. In particular, given ALE's possession of appropriateness, there is a trade-off between representing refinements of the information in a feature structure by feature values and representing them by subtypes. Instead of having three features, as shown above, an *index* type could be declared, for example, with subtypes corresponding to what we know about person, number and gender information:

```
index sub [index_first,index_second,index_third,index_sing,
           index_plur,...].
index_first sub [index_first_sing,index_first_plur,...].
index_first_sing sub [index_first_sing_masc,
                      index_first_sing_fem,...].
```

Many, but not all types can have their features "unfolded" into subtypes in this fashion, so that the resulting signature is logically equivalent. Computationally, this unfolding can actually make unification faster because it avoids the dereference necessary to obtain feature values by compacting those values' representations into the types themselves, where they can be hashed, compiled into efficient bit vector codes, etc. The problem is that this unfolding results in such a potentially vast number of types that the amount of time that it takes to compile a completely unfolded grammar blows up.

Logically, there is very little reason to prefer expanding the features of one eligible type over another, unless some pre-determined maximum number of types is set, which is in turn, an empirically meaningless bound to resort to. What really matters is the frequency with which types are encountered by the unifier at run-time. Those that are more frequent can be unfolded up to some number of types that represent a pre-determined percentage of the total number of unifications that are performed at run-time.

This is a slightly different optimisation in two ways. First, although empirical data must be collected at run-time, those data are used to improve the compiler, not the run-time code. Second, we are not interested in forcing an early failure as such, but rather in securing as large of a speed-up as we can without exceeding certain resource limits.

4. Evaluation

Two of these methods were evaluated on the "Naive HPSG" grammar that is distributed with ALE. Indexing was not tested because of its limited applicability to the large-scale grammars that we have at our disposal (mostly HPSG grammars), which actually place most of the work of rules into procedural attachments rather than into large indices themselves for daughters. This method needs to be tested on large grammars written for a system with a more restricted rule formalism.

The Naive HPSG grammar is an ALE grammar with 162 types and 37 features, with the number of features per type ranging from 2 to 5, with a median across types of 3. It uses almost every piece of functionality that ALE offers, and massively overgenerates semantic representations because of the treatment of quantifier scope in (Pollard and Sag 1994), making it very easy to find computationally intensive parses in a test corpus. It is also a fairly faithful encoding of the grammar for English presented in (Pollard and Sag 1994), and thus could be argued to be a "typical" grammar that one might expect ALE users to be interested in. The results presented here are preliminary in that only this grammar was used, and without an attempt to distinguish the improvement over unification times themselves from the overall improvement witnessed in parsing times. On large-scale parses, feature structure unification typically takes about 58% of total parsing time with this grammar.

Only sister feature reordering, as described above, was tested. On the Naive HPSG grammar, this resulted in an average improvement of 15%

in sentence parsing times, with a maximum of 40%. An analysis of the optimised feature order revealed that most of the improvement was due to the reordering of the features of the type *ne_list*, non-empty lists, and *ne_set*, non-empty sets. Naive HPSG works with many lists and sets that contain only zero, one or two elements. By reordering features in lists so that TAIL is first, one can use the type of that feature's value (either *e_list* or *ne_list*) as an efficient index to measure the length of the list involved, and likewise for sets.

For feature unfolding, the counts obtained for frequency of unification revealed that the most frequently encountered types by the unifier were *index* and *list*-related subtypes. Only *index*'s features were unfolded into subtypes; and this resulted in a 25% to 33% improvement (larger improvement for larger parses) in run-time performance with less than double the compilation time.

5. Future Work

A great deal more needs to be done to investigate the usefulness of statistical optimisations of this kind. In particular, DCND is actually a subcase a more general classification problem, in which a single 'no' answer on any attribute (in the sense used by the classification literature) produces a classification of 'no', e.g., a unification failure for a single appropriate feature results in a global unification failure. The reorderings that we perform are actually adjustments to the order of questions along the 'yes' path of a decision tree.

Normal decision-tree-growing algorithms cannot be used here, however, and not even incremental versions, e.g., (Utgoff 1988), that can cope with learning from new examples on the fly, because we require an algorithm that can cope with learning in spite of having only partial knowledge about new examples. The current method simply makes an assumption of uniformity in the absence of contradictory evidence.

The present method can also lead to thrashing in some cases. For example, when unifying:

$$\begin{bmatrix} \text{FOO} & \\ \text{F} & [0] \\ \text{G} & [0] \end{bmatrix} \quad \sqcup \quad \begin{bmatrix} \text{FOO} & \\ \text{F} & \text{a} \\ \text{G} & \text{b} \end{bmatrix}$$

The bracketed zero indicates an instance of structure-sharing of feature values. If *a* and *b* are not unifiable, then no matter which order of features is chosen, unification on the first will succeed, and unification on the second will fail. The present method will simply continue to swap the orders; and the threshold that triggers reordering can only slow down the thrashing, not prevent it. A reliable way to optimise both threshold settings mentioned here also needs to be found.

References

Aït-Kaci, H. (1991). *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.

Baudinet, M. and D. MacQueen (1985). Tree pattern matching for ml. Extended Abstract.

Carpenter, B. (1992). *The Logic of Typed Feature Structures.* Cambridge.

Carpenter, B. and G. Penn (1996). Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita (Eds.), *Recent Advances in Parsing Technologies.* Kluwer.

Gibson, E. (1992). Empirical category selection in parsing. Unpublished ms., Carnegie Mellon University.

Penn, G. and O. Popescu (1997). Head-driven generation and indexing in ale. In *Proceedings of the ENVGRAM workshop; ACL/EACL.*

Pollard, C. and I. Sag (1994). *Head-driven Phrase Structure Grammar.* Chicago.

Schöter, A. (1992). Term encoding of feature structures.

Utgoff, P. (1988). Id5: An incremental id3. In P. Johnson-Laird (Ed.), *Proceedings of the Fifth International Conference on Machine Learning*, pp. 107–120.