JAUNT: A Constraint Solver for Disjunctive Feature Structures

Hiroshi Maruyama
IBM Research, Tokyo Research Laboratory
maruyama@trl.vnet.ibm.com

Abstract

To represent a combinatorial number of ambiguous interpretations of a natural language sentence efficiently, a "packed" or "factorized" representation is necessary. We propose a representation that comprises a set of explicit value disjunctions and constraints imposed on them. New constraints are successively added for disambiguation, during which local consistencies are maintained by an underlying mechanism. We have developed a constraint solver called JAUNT that embodies this idea. The latest techniques, including *constraint propagation* and *forward checking*, are employed as constraint satisfaction mechanisms. JAUNT also allows an external meta-inference program to intervene in the constraint satisfaction process in order to control the application of the constraints.

1. Introduction

Certain natural language constructs, such as PP-attachment in English, are known to have a combinatorial number of syntactic parses (Church & Patil 1988). For example, sentence (1) has 14 (= Catalan(4)) different parses because of the three consecutive PPs:

> Put the block on the floor on the table in the room. (1)

Representing the set of parses in a compact way and extracting a correct parse by using such knowledge as

> A block cannot be on a floor and on a table at the same time

are keys to a practical natural language system.

The parsing method of Constraint Dependency Grammar (Maruyama 1990) addressed exactly these issues. The essential ideas were

- to represent the set of parses by a *constraint network*, which is composed of a set of explicit value disjunctions and constraints imposed on them,
- to apply constraint propagation in order to keep the constraint network locally consistent, and
- to dynamically add new constraints for disambiguation.

In this paper, we describe a programming tool named JAUNT that embodies the above ideas. JAUNT is a constraint solver for disjunctive feature structures, whose constraint satisfaction mechanisms are *constraint propagation* and *forward checking*. In the next section, we show how various ambiguities are represented in our explicit value disjunction + constraints scheme. The constraint satisfaction algorithms

```
S := {
 [id=0,cat=v,head=put,gr=root,mod=nil],
 [id=1,cat=np,head=block,gr=obj,mod=0],
 [id=2,cat=pp,prep=on,head=floor,
  gr={%loc,postmod%},mod={%0,1%} ],
 [id=3,cat=pp,prep=on,head=table,
  gr={%loc,postmod%},mod={%0,1,2%} ],
 [id=4,cat=pp,prep=in,head=room,
  gr={%loc,postmod%},mod={%0,1,2,3%} ]
};
```

Figure 1: JAUNT representation of sentence (1)

adapted in JAUNT are explained in Section 3. Section 4 describes the use of JAUNT's meta-inference capability. Section 5 concludes the paper.

2. Explicit disjunction + constraints

Let us consider sentence (1). In order to simplify the following discussion, we assume that the sentence is preprocessed as in Figure 1. This preprocessing can be done by a simple context-free grammar that does not determine PP-attachments. In the figure, `[...]` is a feature structure, `{...}` is a list, and `{%...%}` is a disjunction. Thus, the variable `S` represents the (packed) structure of sentence (1) as a list of five components, each of which corresponds to a V, an NP, or a PP. The grammatical relation (`gr=`) and the modifiee (`mod=`) of the three PPs are disjunctions, meaning that one of the values should be selected, but that the correct candidate has not yet been determined. For example, the first PP "on the floor" has `{%0,1%}` as its `mod=` value, which means it can modify either phrase 0 (the verb "put") or phrase 1 (the NP "the block").

Not all the value combinations of the disjunctions are allowed. In the above example, if a PP modifies the main verb, the grammatical relation should be `loc`. In JAUNT, constraints are introduced by `addc` statements. The program fragment (2) applies constraints between the modifiee and the grammatical relation of a PP.

```
for W in S begin
  addc W.cat==pp & S.(W?mod).cat in {pp,np} =>
      W?gr==postmod;
  addc W.cat==pp & S.(W?mod).cat==v =>
      W?gr==loc;
end;                                              (2)
```

Here, dots and question marks are operators for accessing components of lists or feature structures[†]. The

[†]The difference between a dot and a question mark is that a

symbols & (logical and) and => (imply) have their ordinary logical meanings. In general, any first-order logical formula without quantification is allowed as a constraint. The variable W is bound to each V, NP, and PP while the addc statements between begin and end are executed. The first addc statement reads as follows:

> If the category of W is PP and the category of the modifiee of W is either PP or NP, then the grammatical relation of W should be postmod.

The applied constraints are represented implicitly by an internal data structure called a *constraint network* (described later).

In addition, the *projectivity* constraint, that modification links do not crossover, can be programmed as follows:

```
for Y,X in S begin
  addc (Y.id < X.id & X?mod < Y.id) =>
    X?mod <= Y?mod;
end;                                        (3)
```

We have now obtained a packed representation that consists of explicit disjunctions, as in Figure 1, and constraints attached behind them. Each value combination of the disjunctions that globally satisfies the constraints exactly corresponds to one of the 14 parses of sentence (1).

Every context-free parsing algorithm that has a polynomial time bound produces a packed representation of the parsing results (for example, a *chart* in chart parsing (Kaplan 1973), a *parsing matrix* in the CKY method (Younger 1967), and a *shared-packed-forest* in Tomita's algorithm (Tomita 1987)). These representations take advantage of the regularities of syntactic ambiguities in context-free parsing. For example, since it is known that n consecutive PPs have $Catalan(n)$ different parses, it is possible to encode all PP-attachment ambiguities by remembering only n and the position of the PPs (Church & Patil 1982).

However, once we try to extract a single interpretation from these representations, we face a problem, because such regularities may be void when new constraints are introduced for disambiguation. Consider the application of constraint (4):

> A verb cannot have two locatives. (4)

This constraint violates the regularity of the PP-attachment ambiguity and therefore, the CFG-based packed representations mentioned above cannot handle this new information properly without modifying the grammar significantly. In JAUNT, this constraint is applied by a simple addc statement (5).

```
for X,Y in S begin
  addc not(X?mod==Y?mod & S.(X?mod).cat==v &
              X?gr==loc & Y?gr==loc);
end;                                        (5)
```

Formally, it has been proven that Constraint Dependency Grammars, whose rules can be written as

question mark allows a disjunction as its value, whereas a dot does not. The current implementation generates more efficient code for dots than for question marks.

```
S := {
 [id=0,cat=v,head=put,gr=root,mod=nil],
 [id=1,cat=np,head=block,gr=obj,mod=0],
 [id=2,cat=pp,prep=on,head=floor,
  gr={%loc,postmod%},mod={%0,1%} ],
 [id=3,cat=pp,prep=on,head=table,
  gr={%loc,postmod%},mod={%0,2%} ],
 [id=4,cat=pp,prep=in,head=room,
  gr={%loc,postmod%},mod={%0,1,2,3%} ]
};
```

Figure 2: JAUNT representation of sentence (1)

restricted forms of JAUNT program, have a weak generative power strictly greater than that of CFG (Maruyama 1991). This implies that certain types of parsing results can be represented by constraint networks but not by CFG-based representations.

Seo and Simmons (1988) proposed *syntactic graphs* and discussed the advantages of having explicit disjunctions in a packed data structure. Their representation is similar to ours in the sense that they have constraints attached to the explicit disjunctive data structure. However, they do not discuss how to apply disambiguation knowledge in order to reduce the ambiguity effectively. In JAUNT, the underlying constraint satisfaction algorithm removes inconsistent values and keeps the constraint network locally consistent. Consider, for example, the application of the new constraint (6):

> An object cannot be on two distinct objects at the same time. (6)

This constraint is written as follows:

```
for X,Y in S begin
  addc X.prep==on & Y.prep==on &
       X?mod in {pp,np} => X?mod != Y?mod;
end;                                        (7)
```

After this constraint has been evaluated, the mod attribute of the PP "on the table" becomes {%0,2%}, as shown in Figure 2, because the value 1 is locally inconsistent according to the constraints applied so far, and cannot participate in any of the remaining seven readings.[†]

There have been several attempts to incorporate disjunctions in unification-based grammars (e.g. Karttunen 1984). Constraints are introduced by a unification between two disjunctive feature structures. A unification succeeds only if there are combinations of values of the disjunctions that satisfy the equality constraints implied by the unification. In order to clarify the expressive power of feature structures with general disjunctions, Kasper & Rounds (1986) defined a logic-based notation called FML. A formula in FML can be rewritten as an addc statement in JAUNT, and hence, constraints expressed by a unification can also be expressed in JAUNT. In addition, in unification-based grammars, the only basic predicate is equality, and other useful predicates, such as inequalities and set inclusion/membership, are difficult to represent. In

[†]If the second PP "on the table" modifies the NP "the block," the first PP "on the floor" has no legal modifiees.

JAUNT, inequalities and set operations are built-in, and user-defined predicates are also allowed.

3. Constraint-satisfaction algorithm

Since every disjunction in a JAUNT program has a finite number of choices, its satisfiability problem can be formulated as a constraint satisfaction problem over a finite domain (sometimes called a *consistent-labeling problem* (Montanari 1974)). Much effort has been devoted to developing efficient algorithms for this problem.

Two such algorithms are employed in JAUNT. One is the *constraint propagation* algorithm (Mackworth 1977), which is activated when a new constraint is added by addc statements. The constraint propagation algorithm runs in polynomial time, and eliminates locally inconsistent values from the choice points and propagates the results to the neighboring constraints. The constraint propagation algorithm usually reduces the size of the search space significantly.

The other algorithm used in JAUNT is the *forward-checking* algorithm (Haralick & Elliott 1980), which is triggered by the execution of a special find statement. It is essentially a back-tracking algorithm, but it prunes unpromising branches whenever temporal choices are made, thus significantly reducing the size of the remaining search space.

This section describes in detail the constraint propagation algorithm used in JAUNT. Readers are referred to Hentenryck (1989) for the forward-checking algorithm.

3.1 Internal representation of constraints

Before describing the algorithm in detail, let us explain the internal representation of the constraints. In a compiled module of a JAUNT program, a disjunction is represented by a data structure called a *Choice Point (CP)*. A CP maintains a list of possible values (called a *domain*) at the time of program execution. When a new constraint is added by a addc statement, the constraint is represented internally as a *constraint matrix*. For example, assume that W is bound to

```
[gr={%loc,postmod%}, mod={%0,1%}].
```

W?gr and W?mod are represented internally as CPs whose domain size is two. Then, when the constraints (2) are evaluated, a new two-dimensional constraint matrix is created between the two CPs, as shown in Figure 3.

Each dimension of the constraint matrix corresponds to a CP. The elements indicate whether the particular combination of the CP values is legal (1) or illegal (0). For example, W?gr=loc and W?mod=0 satisfies the constraint and hence the corresponding element in the matrix is 1.

If another addc statement is then executed declaring that the value combination of W?gr=postmod and W?mod=1 is illegal, the corresponding element in the matrix is changed to 0, yielding the matrix shown in Figure 4.

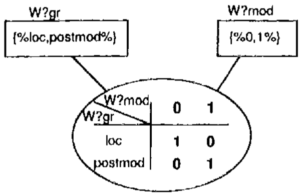

Figure 3: Constraint matrix

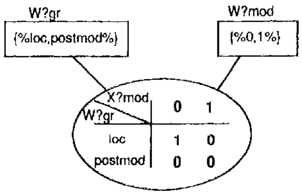

Figure 4: Updated constraint matrix

Suppose that the execution of an addc statement referring to n different CPs $X_1, X_2, ..., X_n$ reveals that the value combination $< x_1, x_2, ..., x_n >$ is illegal. JAUNT first locates an n-dimensional constraint matrix connected to $X_1, X_2, ..., X_n$, and set its element corresponding to the value combination $< x_1, x_2, ..., x_n >$ to 0. If there is no such constraint matrix, JAUNT creates a new one whose elements are all 1 except for the element of $< x_1, x_2, ..., x_n >$ that is set to 0.

3.2 Constraint propagation

The basic idea of constraint propagation is to remove locally inconsistent values from the choice points and to reduce their domain size before a back-tracking search is performed.

In the example above, let us consider the row of W?gr=postmod in the constraint matrix. When W?gr=postmod, the elements of the matrix are zero, whatever value W?mod takes. This means that there are no global solutions with W?gr=postmod, and therefore this value can be safely removed from the domain of the CP W?gr. Similarly, W?mod=1 can be removed from the domain of the CP W?mod.

In general, when a particular row or column (or plane or hyperplane, if the dimension is greater than two) contains all zero elements, the corresponding value x_i of CP X can never participate in a solution (see Figure 5). Therefore, x_i can be eliminated from the domain of X. Whenever a constraint matrix is updated, JAUNT searches for a hyperplane whose elements are all zero and removes the corresponding value from its domain. This may update other constraint matrices connected to the CP, and may cause values in other CPs to be eliminated. Thus, updates are propagated over the network of constraints until the entire network reaches a stable state.

For every hyperplane in a constraint matrix, JAUNT

$X \setminus Y$				$\ldots$	
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdots$	$\cdot$
x_i	0	0	0	...	0
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdots$	$\cdot$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdots$	$\cdot$

Figure 5: Locally inconsistent value x_i

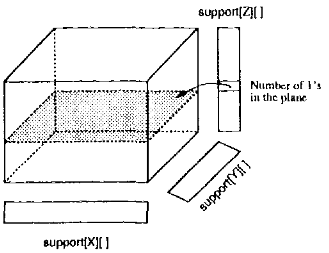

Figure 6: Support

keeps the current number of 1's on that plane, called the *support* (see Figure 6). When a certain element of a constraint matrix appears to be inconsistent as a result of the evaluation of addc statement, the corresponding support in each dimension is decremented. When a value in a CP is removed by constraint propagation, the corresponding hyperplane of every constraint matrix connected to the CP is removed, and the result is reflected in all the support values in the matrix. This algorithm is a natural extension of Mohr and Henderson's arc-consistency algorithm (Mohr & Henderson 1986) for allowing n-ary constraints.

The computational complexity of our constraint propagation algorithm is bounded by $O(e|M|)$, where $|M|$ is the size of the constraint matrices and e is the number of the constraint matrices, because at least one element in some matrix is changed to 0 from 1 for every iteration of constraint propagation. If the constraints are local, that is, if the arity of each constraint is bounded by a small integer, this time bound is a polynomial of the number of disjunctions.

Our algorithm tries to maintain local consistency in the sense that it considers only one constraint matrix at a time. This is a generalization of the notion called *arc consistency* (Mackworth 1977) or *pair-wise consistency*, and is equivalent to the first two steps of Kasper's (1987) *successive approximation*. Algorithms for achieving more global consistency by looking at multiple constraint matrices are possible, but as Carter (1990) argues in his paper on the experimental Propane parser, once pair-wise consistencies have been achieved, performing a backtrack search is usually more efficient than using higher-level consistency algorithms. In JAUNT, a forward-checking algorithm, which is far better than the traditional backtracking algorithms (Haralick & Elliot 1980), is provided for generating global solutions, if necessary, although the intended use of JAUNT is to combine constraint propagation with the meta-inference described in the next section, rather than to perform a search.

There have been attempts to formulate natural language processing as a constraint satisfaction problem with broader domains (for example, the Herbrand domain). CIL (Mukai 1988) and cu-Prolog (Tsuda, Hasida & Sirai 1989) are examples of such attempts. There is a trade-off between the expressive power and the computational complexity, and we argue that finite domains have sufficient expressive power while retaining the computational efficiency implied by the algorithms described above.

4. Meta-inference

A consistent-labeling problem may or may not have a solution. If it has one, it is most probable that there are multiple solutions. In fact, if the given constraints are not 'tight' enough to narrow down the number of solutions to one or a few, the problem may have an exponential number of solutions. This situation is common in natural language processing. Strict grammars cause analysis failures for grammatical sentences; on the other hand, loose grammars produce a combinatorially explosive number of parse trees for certain types of sentence. To avoid this situation, constraints should be dynamically added and removed according to the size of the solution space. In other words, a constraint solver should be provided with a means of watching its own inference process and changing its strategy according to the observation.

To support the meta-inference capability, JAUNT provides the following built-in functions:

1. `inconsistentp()` ... Non-NULL when JAUNT detects inconsistencies between constraints
2. `saveState()` ... Save the current status of constraint satisfaction
3. `loadState()` ... Restore the saved status of constraint satisfaction.

In JAUNT, the state of the constraint-satisfaction process is defined as the set of all choice points and all constraint matrices. Other statuses such as global and local variables, the program counter, and the control stack are not saved, so applications of constraints can be undone without disturbing the control flow.

Meta-inference is sometimes performed in an external module. JAUNT has inter-process communication primitives based on UNIX sockets. With these meta-inference capabilities, an independent inference process using external knowledge can monitor and intervene in a JAUNT program. If it detects an inconsistency, it instructs the JAUNT program to go back to the previous inference state and try another set of constraints; if it finds that the solution space is not small enough, it may give new constraints from its own knowledge source. By separating the meta-inference module from the object-level JAUNT program, modularity of knowledge is achieved.

As an application of the meta-inference capability, let us describe the interactive Japanese parser of the Japanese-to-English machine translation system JETS (Maruyama, Watanabe, & Ogino 1990). The system structure is shown in Figure 7. The morphological

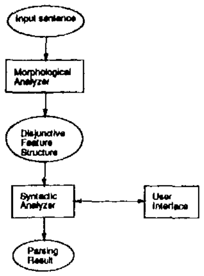

Figure 7: Analysis part of JETS

```
{ [word_id=0,
   string="ANATA",
   modifiee={%1,2,3,4%},
   lex={% [part_of_speech=pronoun, sf={hum}],
          [part_of_speech=noun, sf={loc}]
   %},
   .
   .
}
```

Figure 8: Input feature structure

analyzer analyzes an input sentence using a type-3 grammar and creates a feature structure that contains disjunctions for lexical and attachment ambiguities (Figure 8). The syntactic analysis program written in JAUNT applies grammatical constraints based on Constraint Dependency Grammar to these choice points and sends the result to a user-interface running on a separate machine. The ambiguous choice points (those with domain size> 1) are highlighted on the screen, and the end user can select an appropriate value for some of them. This information is sent back to the JAUNT program through the inter-process communication channel and applied in the form of new constraints. This iteration is written in JAUNT as follows:

```
Uif := open(ClientName,"socket");
while true begin
   send(Uif,S);
   X := read(Uif);
   if X==goAhead then break;
   saveState();
   addc S.(X.id)?mod==X.mod;
   if inconsistentp() then begin
      send(Uif,"inconsistency detected");
      loadState();
   end;
end;
```

Thus, in JETS, the end user acts as an external knowledge source to guide the inference process of the program.

SHALT2, an experimental English-to-Japanese machine translation system currently being developed at IBM's Tokyo Research Laboratory, has a similar system structure (Nagao 1990). Instead of user interaction, an external example base built from an existing corpus is used for resolving attachment ambiguities in SHALT2. Thus, clear modularization of general syntactic/semantic knowledge from domain-dependent example-based knowledge is achieved.

5. Conclusion

We have described a constraint solver for efficiently processing ambiguities in natural language sentences. Disambiguation is done by dynamically adding new constraints while the constraint satisfaction algorithm maintains local consistency. The system is actually implemented and used in two machine translation systems.

References

1. Carter, D., 1990, "Efficient Disjunctive Unification for Bottom-Up Parsing," *COLING '90.*
2. Church, K. and Patil, R., 1982 "Coping with Syntactic Ambiguity, or How to Put the Block in the Box on the Table," *American J. of Computational Linguistics 8.*
3. Haralick, M. and Elliott, G. L., 1980, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence 14.*
4. Hentenryck, P. V., 1989, *Constraint Satisfaction in Logic Programming,* MIT Press.
5. Karttunen, L., 1984, "Features and Values," *COLING '84.*
6. Kasper, R. T., 1987, "A Unification Method for Disjunctive Feature Descriptions," *25th ACL Annual Meeting.*
7. Kasper, R. T., and Rounds, W. C., 1986, "A Logical Semantics for Feature Structures," *24th ACL Annual Meeting.*
8. Mackworth, A. K., 1977, "Consistency in Networks of Relation," *Artificial Intelligence 8.*
9. Maruyama, H., 1990, "Structural Disambiguation with Constraint Propagation," *28th ACL Annual Meeting.*
10. Maruyama, H., 1991, "Constraint Dependency Grammar and Its Weak Generative Capacity," *Advances in Software Science and Technology 3.*
11. Maruyama, H., Watanabe, H., and Ogino, S., 1990, "An Interactive Japanese Parser for Machine Translation," *COLING '90.*
12. Montanari, U., 1974, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Science 7.*
13. Mohr, R. and Henderson, T., 1986, "Arc and Path Consistency Revisited," *Artificial Intelligence 28.*
14. Mukai, K., 1988, "Partially Specified Term in Logic Programming for Linguistic Analysis," *International Conference on Fifth Generation Computer Systems,* Tokyo.
15. Nagao, K., 1990, "Constraints and Preferences: Integrating Grammatical and Semantic Knowledge for Structural Disambiguation," *Pacific Rim International Conference on AI,* Nagoya.
16. Seo, J. and Simmons, R. 1988, "Syntactic Graphs: a Representation for the Union of All Ambiguous Parse Trees," *Computational Linguistics 15.*
17. Tsuda, H., Hasida, K., and Sirai, H., 1989, "JPSG Parser on Constraint Logic Programming," *4th ACL European Chapter.*
18. Younger, D. H., 1967, "Recognition and Parsing of Context-Free Languages in time n^3," *Information and Control 10.*