Dynamic Dependency Parsing

Michael Daum
Natural Language Systems Group
Department of Computer Science
University of Hamburg
micha@nats.informatik.uni-hamburg.de

Abstract

The inherent robustness of a system might be an important prerequisite for an incremental parsing model to the effect that grammaticality requirements on full sentences may be suspended or allowed to be violated transiently. However, we present additional means that allow the grammarian to model prefix-analyses by altering a grammar for non-incremental parsing in a controlled way. This is done by introducing underspecified dependency edges that model the expected relation between already seen and yet unseen words during parsing. Thus the basic framework of weighted constraint dependency parsing is extended by the notion of dynamic dependency parsing.

1 Introduction

In an incremental mode of operation, a parser works on a prefix of a prolonging utterance, trying to compute prefix-analyses while having to cope with a growing computational effort. This situation gives rise at least to the following questions:

(1) Which provisions can be made to accept prefix-analyses transiently given a model of language that describes *complete* sentences?

(2) How shall prefix-analyses look like?

(3) How can the complexity of incremental parsing be bounded?

We will introduce underspecified dependency edges, called *nonspec* dependency edges, to the framework of *weighted constraint dependency grammar* (WCDG) (Schröder, 2002). These are used to encode an expected function of a word already seen but not yet integrated into the rest of the parse tree during incremental parsing.

In WCDG, parse trees are annotated by constraint violations that pinpoint deviations from grammatical requirements or preferences. Hence weighted constraints are a means to describe a graded grammaticality discretion by describing the inherent 'costs' of accepting an imperfect parse tree. Thus parsing follows principles of economy when repairing constraint violations as long as reducing costs any further is justified by its effort.

The following sections revise the basic ideas of applying constraint optimization to natural language parsing and extend it to dynamic dependency parsing.

2 From static to dynamic constraint satisfaction

We begin by describing the standard constraint satisfaction problem (CSP), then extend it in two different directions commonly found in the literature: (a) to constraint optimization problems (COP) and (b) to dynamic constraint satisfaction problems (DynCSP) aggregating both to dynamic constraint optimization problems (DynCOP) which is motivated by our current application to incremental parsing[1].

2.1 Constraint Satisfaction

Constraint satisfaction is defined as being the problem of finding consistent values for a fixed set of variables given all constraints between those values. Formally, a *constraint satisfaction problem* (CSP) can be viewed as a triple (X, D, C) where $X = \{x_1, \ldots, x_n\}$ is a finite set of variables with respective *domains* $D = \{D_1, \ldots, D_n\}$, and a set of *constraints* $C = \{C_1, \ldots, C_t\}$. A constraint C_i is defined as a relation defined on a subset of variables, called

[1]Note that we don't use the common abbreviations for *dynamic constraint satisfaction problems* DCSP in favor of DynCSP in order to distinguish if from *distributed constraint satisfaction problems* which are called DCSPs also. Likewise we use DynCOP instead of DCOP, the latter of which is commonly known as *distributed constraint optimization problems*.

the *scope*, restricting their simultaneous assignment. Constraints defined on one variable are called *unary*; constraints on two variables are *binary*. We call unary and binary constraints *local* constraints as their scope is very restricted. Constraints of wider scope are classified *non-local*. Especially those involving a full scope over all variables are called *context* constraints. The 'local knowledge' of a CSP is encoded in a *constraint network* (CN) consisting of nodes bundling all values of a variable consistent with all unary constraints. The edges of a CN depict binary constraints between the connected variables. So a CN is a compact representation (of a superset) of all possible instantiations. A solution of a CSP is a complete instantiation of variables $\langle x_1, \ldots, x_n \rangle$ with values $\langle d_{i_1}, \ldots, d_{i_n} \rangle$ with $d_{i_k} \in D_k$ found in a CN that is consistent with all constraints.

Principles of processing CSPs have been developed in (Montanari, 1974), (Waltz, 1975) and (Mackworth, 1977).

2.2 Constraint Optimization

In many problem cases no complete instantiation exists that satisfies all constraints: either we get stuck by solving only a part of the problem or constraints need to be considered defeasible for a certain penalty. Thus finding a solution becomes a *constraint optimization problem* (COP). A COP is denoted as a quadruple (X, D, C, f), where (X, D, C) is a CSP and f is a cost function on (partial) variable instantiations. f might be computed by multiplying the penalties of all violated constraints. A solution of a COP is a complete instantiation, where $f(\langle d_{i_1}, \ldots, d_{i_n} \rangle)$ is optimal. This term becomes zero if the penalty of at least one violated constraint is zero. These constraints are called *hard*, those with a penalty greater zero are called *soft*.

An more precise formulation of COPs (also called *partial constraint satisfaction problems*), can be found in (Freuder and Wallace, 1989).

2.3 Dynamic Constraint Satisfaction

The traditional CSP and COP framework is only applicable to static problems, where the number of variables, the values in their domains and the constraints are all known in advance. In a dynamically changing environment these assumptions don't hold any more as new variables, new values or new constraints become available over time. A *dynamic constraint satisfaction problem* (DynCSP) is construed as a series of CSPs $P_0, P_1, \ldots$ that change periodically over time by loss of gain of values, variables or constraints ($P_{i+1} = P_i + \Delta_{P_{i+1}}$). For each problem change $\Delta_{P_{i+1}}$ we try to find a solution change $\Delta_{S_{i+1}}$ such that $S_{i+1} = S_i + \Delta_{S_{i+1}}$ is a solution to P_{i+1}. The legitimate hope is that this is more efficient than solving P_{i+1} the naive way from scratch whenever things change.

This notation is consistent with previous ones found in in (Dechter and Dechter, 1988) and (Wirén, 1993).

2.4 Dynamic Constraint Optimization

Most notions of DynCSPs in the literature are an extension of the classical CSP that use hard constraints exclusively. To model the aimed application of incremental parsing however, we still like to use weighted constraints. Therefore we define *dynamic constraint optimization problems* (DynCOP) the same way DynCSPs were defined on the basis of CSPs as a series of COPs $P_0, P_1, \ldots$ that change over time. In addition to changing variables, values and constraints we are concerned with changes of the cost function as well. In particular, variable instantiations evaluated formerly might now be judged differently. As this could entail serious computational problems we try to keep changes in the cost function monotonic, that is re-evaluation shall only give lower penalties than before, i.e. instantiations that become inconsistent once don't get consistent later on again.

3 Basic Dependency Parsing

Using constraint satisfaction techniques for natural language parsing was introduced first in (Maruyama, 1990) by defining a *constraint dependency grammar* (CDG) that maps nicely on the notion of a CSP. A CDG is a quadruple (Σ, R, L, C), where Σ is a lexicon of known words, R is a set of *roles* of a word. A role represents a level of language like 'SYN' or 'SEM'. L is a set of labels for each role (e.g. {'SUBJ', 'OBJ'}, {'AGENT','PATIENT'}), and C is a *constraint grammar* consisting of atomic logical formulas. Now, the only thing that is left in order to match a CDGs to a CSPs is to define variables and their possible values. For each word of an utterance and for each role we allocate one variable that can take values of the form $e_{i,j} = \langle r, w_i, l, w_j \rangle$ with $r \in R$, $w_i, w_j \in \Sigma$ and $l \in L$. $e_{i,j}$ is called the *dependency edge* between word form w_i and w_j labeled with l on the description level r. A dependency edge of

the form $e_{i,root}$ is called the *root edge*. Hence a *dependency tree* of an utterance of length n is a set of dependency edges $s = \{e_{i,j} \mid i \in \{1,\ldots,n\}\ ,\ j \in \{1,\ldots,n\} \cup \{root\}\ ,\ i \neq j\}$.

From this point on parsing natural language has become a matter of constraint processing as can be found in the CSP literature (Dechter, 2001).

4 Weighted Dependency Parsing

In (Schröder, 2002) the foundations of dependency parsing have been carried over to COPs using *weighted constraint dependency grammars* (WCDG), a framework to model language using all-quantified logical formulas on dependency structures. Penalties for constraint violations aren't necessarily static once, but can be lexicalized or computed arithmetically on the basis of the structure under consideration. The following constraints are rather typical once restricting the properties of subject edges:

```
{X:SYN} : SUBJ-init : 0.0 :
  X.label = SUBJ ->
  ( X@cat = NN | X@cat = NE |
    X@cat = FM | X@cat = PPER ) &
  X^cat = VVFIN;

{X:SYN} : SUBJ-dist : 2.9 / X.length :
  X.label = SUBJ -> X.length < 3;
```

Both constraints have a scope of one dependency edge on the syntax level (`{X:SYN}`). The constraint `SUBJ-init` is a hard constraint stating that every dependency edge labeled `SUBJ` shall have a nominal modifier and a finite verb as its modifiee. The second constraint `SUBJ-dist` is a soft one, such as every edge with label `SUBJ` attached more than two words away induces a penalty calculated by the term `2.9 / X.length`. Note, that the maximal edge length in `SUBJ-dist` is quite arbitrary and should be extracted from a corpus automatically as well as the grade of increasing penalization. A realistic grammar consists of about 500 such handwritten constraints like the currently developed grammar for German (Daum et al., 2003).

The notation used for constraints in this paper is expressing valid formulas interpretable by the WCDG constraint system. The following definitions explain some of the primitives that are part of the constraint language:

- `X` is a variable for a dependency edge of the form $e_{i,j} = \langle r, w_i, l, w_j \rangle$,

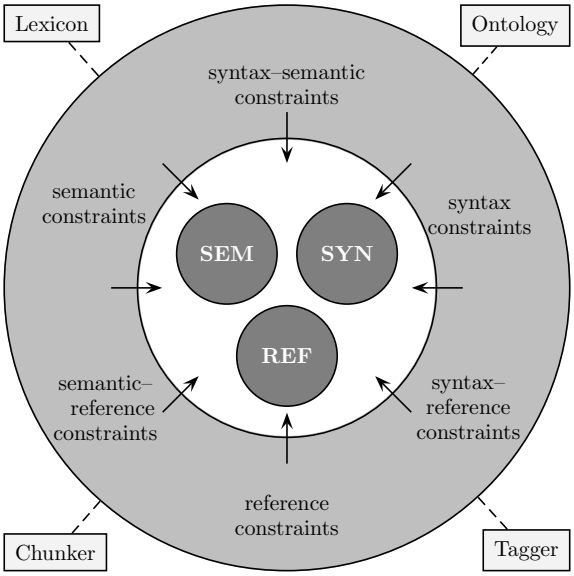

Figure 1: Architecture of WCDG

- `X@word` (`X^word`) refers to the word form $w_i \in \Sigma$ ($w_j \in \Sigma$)
- `X@id` (`X@id`) refers to the position i (j)
- `X.label` refers to the label $l \in L$
- `X@cat` (`X^cat`) refers to the POS-tag of the modifier (modifiee)
- `root(X^id)` $\equiv$ `true` iff $w_j = root$
- `X.length` is defined as $|i - j|$.

A complete definition can be found in (Schröder et al., 1999).

Figure (1) outlines the overall architecture of the system consisting of a lexicon component, ontologies and other external shallow parsing components, all of which are accessed via constraints affecting the internal constraint network as far as variables are in their scope. While a static parsing model injects all variables into the ring in Figure (1) once and then waits for all constraints to let the variable instantiations settle in a state of equilibrium, a dynamic optimization process will add and remove variables from the current scope repeatedly.

5 Dynamic Dependency Parsing

As indicated, the basic parsing model in WCDG is a two stage process: first building up a constraint network given an utterance and second constructing an optimal dependency parse. In a dynamic system like an incremental dependency parser these two steps are repeated in a loop while consuming all bits from the input that

complete a sentence over time. In principle, the problem of converting the static parsing model into a dynamic one should only be a question of repetitive updating the constraint network in a subtle way. Additionally, information about the internal state of the 'constraint optimizer' itself, which is not stored in the constraint net, shall not get lost during consecutive iterations as it (a) might participate in the update heuristics of the first phase and (b) the parsing effort during *all* previous loops might affect the current computation substantially. We will come back to this argument in Section 8.

Basically, changes to the constraint network are only allowed after the parser has emitted a parse tree. This is acceptable if the parser itself is interruptible providing the best parse tree found so far. An interrupt may occur either from 'outside' or from 'inside' by the parser itself taking into account the number of pending new words not yet added. So it either may integrate a new word as soon as it arrives or wait until further hypotheses have been checked. As transformation based parsing has strong anytime properties, these heuristics can be implemented as part of a termination criterion between increments easily.

6 Modeling expectations using *nonspec*

6.1 Motivation

Analyzing sentence prefixes with a static parser, that i.e. is not aware of the sentence being a prefix, will yield at least a penalty for a fragmentary representation. To get such a result at all, the parser must allow partial parse trees. The constraints `S-init` and `frag` illustrate modeling of normal and fragmentary dependency trees.

```
{X:SYN} : S-init : 0.0 :
  X.label = S -> root(X^id) &
    (X@cat = VVFIN | X@cat = VMFIN |
     X@cat = VAFIN | ... );

{X:SYN} : frag : 0.001 :
  root(X^id) -> X.label = S;
```

Constraint `S-init` restricts all edges with label `S` to be finite verbs pointing to *root*. But if some dependency edge is pointing to *root* and is *not* labeled with `S` then constraint `frag` is violated and induces a penalty of `0.001`. So every fragment in sentence (2a) that can not be integrated into the rest of the dependency tree will increase the penalty of the structure by three orders of magnitude. A constraint optimizer will try to avoid an analysis with an overall penalty of at least 1^{-12} and will search for another structure better than that. Modeling language in a way that (2a) in fact turns out as the optimal solution is therefore difficult. Moreover, the computational effort could be avoided if a partial tree is promised to be integrated *later* with fewer costs.

The only way to prevent a violation of `frag` in WCDG is either by temporarily switching it off completely or, preferably, by replacing the *root* attachment with a *nonspec* dependency as shown in (2b), thereby preventing the prerequisites of `frag` in the first place while remaining relevant for 'normal' dependency edges.

A prefix-analysis like (2a) might turn out to be cognitively implausible as well, as humans expect a proper NP-head to appear as long as no other evidence forces an adjective to be nominalized. Such a thesis can be modeled using *nonspec* dependency edges.

6.2 Definition

We now extend the original definition of WCDG, so that a dependency tree is devised as $s = \{e_{i,j} \mid i \in \{1,\ldots,n\} \cup \{*\}, j \in \{1,\ldots,n\} \cup \{root, *\}, i \neq j\}$. We will use the notation w_* to denote any unseen word. A dependency edge modifying w_* is written as $e_{i,*}$, and an edge of the form $e_{*,i}$ denotes a dependency edge of w_* modifying an already seen word. $e_{i,*}$ and $e_{*,i}$ are called *nonspec* dependency edges.

Selective changes to the semantics of the constraint language have been made to accomplish *nonspec* dependency edges. So given two edges $e_{i_1,i_2} = \langle r, w_{i_1}, l', w_{i_2}\rangle$ and $e_{j_1,j_2} =$

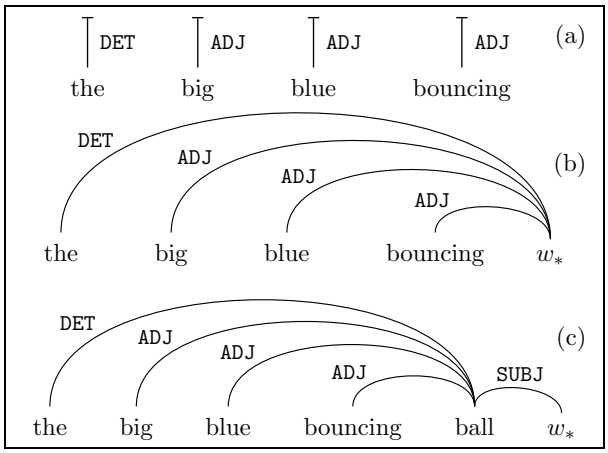

Figure 2: Example sentence prefix

$\langle r, w_{j_1}, l'', w_{j_2} \rangle$ with `X` $\equiv e_{i_1,i_2}$ and `Y` $\equiv e_{j_1,j_2}$:

- `X^id = Y^id` $\equiv$ `false` iff $w_{i_2} \neq w_{j_2} \in \Sigma \vee w_{i_2} = w_{j_2} = w_*$, and `true` otherwise
- `X.length` $\equiv |i_1 - i_2|$ iff $w_{i_1}, w_{i_2} \in \Sigma$, and $n+1$ iff $w_{i_2} = w_*$, (n: length of the current sentence prefix)
- `X^cat` = $\langle POS-tag \rangle \equiv$ `false` iff $w_{i_2} = w_*$
- `nonspec(X^id)` $\equiv$ `true` iff $w_{i_2} = w_*$
- `spec(X^id)` $\equiv$ `true` iff $w_{i_2} \in \Sigma$

6.3 Properties

Although every *nonspec* dependency in Figure (2b) points to the same word w_*, two *nonspec* dependency edges are not taken to be connected at the top (`X^id = Y^id` $\equiv$ `false`) as we don't know yet whether w_i and w_j will be modifying the same word in the future.

In general, the above extension to the constraint language is reasonable enough to fit into the notion of static parsing, that is a grammar tailored for incremental parsing can still be used for static parsing. An unpleasant consequence of *nonspec* is, that more error-cases might occur in an already existing constraint grammar for static parsing that was not written with *nonspec* dependency edges in mind. Therefore we introduced guard-predicates `nonspec()` and `spec()` that complete those guard-predicates already part of the language (e.g. `root()` and `exists()`). These must be added by the grammarian to prevent logical error-cases in a constraint formula.

Looking back at the constraints we've discussed so far, the constraints `SUBJ-init` and `S-init` have to be adjusted to become *nonspec*-aware because referring to the POS-tag is not possible if the dependency edge under consideration is of the form $e_{i,*}$ or $e_{*,i}$. Thus a prefix-analysis like (2c) is inducing a hard violation of `SUBJ-init`. We have to rewrite `SUBJ-init` to allow (2c) as follows:

```
{X:SYN} : SUBJ-init : 0.0 :
  X.label = SUBJ ->
  ( nonspec(X@id) |
    X@cat = NN | X@cat = NE |
    X@cat = FM | X@cat = PPER) &
  ( nonspec(X^id) | X^cat = VVFIN );
```

When all constraints have been checked for logical errors due to a possible *nonspec* dependency edge, performance of the modified grammar will not have changed for static parsing but will accept *nonspec* dependency edges.

Using the `nonspec()` predicate, we are able to write constraints that are triggered by *nonspec* edges only being not pertinent for 'normal' edges. For example we like to penalize *nonspec* dependency edges the older they become during the incremental course and thereby allow a cheaper structure to take over seamlessly. This can easily be achieved with a constraint like `nonspec-dist`, similar to `SUBJ-dist`:

```
{X:SYN} : nonspec-dist : 1.9 / X.length :
  nonspec(X^id) -> X.length < 2;
```

The effect of `nonspec-dist` is, that a certain amount of penalty is caused by $\langle$`SYN`, the, `DET`, $w_*\rangle$ and $\langle$`SYN`, big, `ADJ`, $w_*\rangle$ in (2b). Figure (2c) illustrates the desired prefix-analysis in the next loop when *nonspec* edges become pricey due to their increased attachment length. In a real-life constraint grammar (2c) will be optimal basically because the head of the NP occurred, therefore overruling every alternative *nonspec* dependency edges that crosses the head. The latter alternative structure will either cause a projectivity violation with all other non-head components of the NP that are still linked to the head or cause an alternative head to be elected when becoming available.

7 Dynamic Constraint Networks

nonspec dependency edges play an important role when updating a constraint network to reflect the problem change Δ_{P_i}. Maintaining the constraint network in the first phase is crucial for the overall performance as a more sophisticated strategy to prune edges might compensate computational effort in the second phase.

Figure (3) illustrates a sentence of three words being processed one word w_i per time-point t_i as follows:

1. for each edge e of the form $e_{j,*}$ or $e_{*,j}$, ($j < i$) recompute penalty $f(\langle e \rangle)$. If its penalty drops below α, then remove e. Otherwise derive edge e' on the basis of e
2. add new edges $e_{i,*}$ and $e_{*,i}$ to the CN as far as $f(\langle e_{i,*} \rangle) < \alpha$ and $f(\langle e_{*,i} \rangle) < \alpha$
3. remove each edge e from the CN if it's local penalty is lower than the penalty of the best parse so far.

The parameter α is a penalty threshold that determines the amount of *nonspec* edges being pruned. Any remaining *nonspec* edge indicates where the constraint network remains extensible

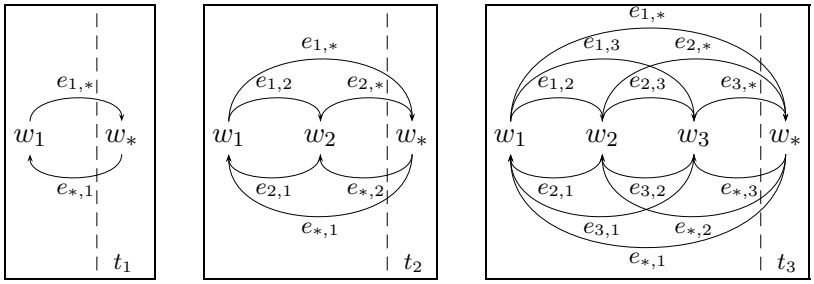

Figure 3: Incremental update of a constraint network

and provides an upper estimate of any future edge derived from it. This holds only if some prerequisites of monotony are guaranteed:

- The penalty of a parse will always be lower than each of the penalties on its dependency edges (guaranteed by the multiplicative cost function).
- Each *nonspec* edge must have a penalty that is an upper boundary of each dependency edge that will be derived from it:
  $f(\langle e_{*,i_1} \rangle) >= f(\langle e_{i_2,i_1} \rangle)$ and
  $f(\langle e_{i_1,*} \rangle) >= f(\langle e_{i_1,i_2} \rangle)$ with $(i_1 < i_2)$.
  Only then will pruning of *nonspec* dependency edges be correct.
- As a consequence the overall penalties of prefix-analyses degrade monotonically over time: $f(s_i) >= f(s_{i+1})$

Note, that the given strategy to update the constraint network does *not* take the structure of the previous prefix-analysis into account but only works on the basis of the complete constraint network. Nevertheless, the previous parse tree is used as a starting point for the next optimization step, so that near-by parse trees will be constructed within a few transformation steps using the alternatives licensed by the constraint network.

8 The Optimizer

So far we discussed the first phase of a dynamic dependency parser building up a series of problems $P_0, P_1, \ldots$ changing P_i using $\Delta_{P_{i+1}}$ in terms of maintaining a dynamic constraint network. In the second phase 'the optimizer' tries to accommodate to those changes by constructing S_{i+1} on the basis of S_i and P_{i+1}.

WCDG offers a decent set of methods to compute the second phase, one of which implements a *guided local search* (Daum and Menzel, 2002).

The key idea of GLS is to add a heuristics sitting on top of a local search procedure by introducing weights for each possible dependency edge in the constraint network. Initially being zero, weights are increased steadily if a local search settles in a local optimum. By augmenting the cost function f with these extra weights, further transformations are initiated along the gradient of f. Thus every weight of a dependency edge resembles an custom-tailored constraint whose penalty is learned during search.

The question now to be asked is, how weights acquired during the incremental course of parsing influence GLS. The interesting property is that the weights of dependency edges integrated earlier will always tend to be higher than weights of most recently introduced dependency edges as a matter of saturation. Thus keeping old weights will prevent GLS from changing old dependency edges and encourage transforming newer dependency edges first. Old dependency edges will not be transformed until more recent constraint violations have been removed or old structures are strongly deprecated recently. This is a desirable behavior as it stabilizes former dependency structures with no extra provisions to the base mechanism. Transformations will be focused on the most recently added dependency edges. This approach is comparable to a simulated annealing heuristics where transformations are getting more infrequent due to a declining 'temperature'.

Another very successful implementation of 'the optimizer' in WCDG is called *Frobbing* (Foth et al., 2000) which is a transformation based parsing technique similar to taboo search. One interesting feature of Frobbing is its ability to estimate an upper boundary of the penalty of any structure using a certain dependency edge and a certain word form. In an incremental parsing mode the penalty limit of a *nonspec* dependency edge will then be an estimate of any structure derived from it and thereby provide a good heuristics to prune *nonspec* edges falling beyond α during the maintenance of the con-

straint network.

9 Conclusion

Incremental parsing using weighted constraint optimization has been classified as a special case of dynamic dependency parsing.

The idea of *nonspec* dependency edges has been described as a means of expressing expectations during the incremental process. We have argued that (a) *nonspec* dependency edges are more adequate to model prefix-analyses and (b) offer a computational advantage compared to a parser that models the special situation of a sentence prefix only by means of violated constraints.

While completing the notion of dynamic dependency parsing, we assessed the consequences of an incremental parsing mode to the most commonly used optimization methods used in WCDG.

Further research will need to add the notion of DynCSP to the WCDG system as well as an adaption and completion of an existing constraint grammar. This will allow an in-depth evaluation of dynamic dependency parsing with and without *nonspec* dependency edges given the optimization methods currently available. Experiments will be conducted to acquire parsing times per increment that are then compared to human reading times.

Acknowledgments

This research has been partially supported by Deutsche Forschungsgemeinschaft under grant Me 1472/4-1.

References

Michael Daum and Wolfgang Menzel. 2002. Parsing natural language using guided local search. In F. van Harmelen, editor, *Proc. 15th European Conference on Artificial Intelligence*, Amsterdam. IOS Press.

Michael Daum, Kilian Foth, and Wolfgang Menzel. 2003. Constraint based integration of deep and shallow parsing techniques. In *Proceedings 11th Conference of the European Chapter of the ACL*, Budapest, Hungary.

Rina Dechter and Avi Dechter. 1988. Belief Maintenance in Dynamic Constraint Networks. In *7th Annual Conference of the American Association of Artificial Intelligence*, pages 37–42.

Rina Dechter. 2001. *Constraint Processing*. Morgan Kaufmann, September.

Kilian Foth, Wolfgang Menzel, and Ingo Schröder. 2000. A transformation-based parsing technique with anytime properties. In *Proc. 4th International Workshop on Parsing Technologies*, pages 89–100, Trento, Italy.

Eugene C. Freuder and Richard J. Wallace. 1989. Partial constraint satisfaction. In *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, volume 58, pages 278–283, Detroit, Michigan, USA.

A. K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence*. 8:99-118.

Hiroshi Maruyama. 1990. Structure disambiguation with constraint propagation. In *Proc. the 28th Annual Meeting of the ACL*, pages 31–38, Pittsburgh.

U. Montanari. 1974. Networks of constraints: Fundamental properties and applications to picture processing. Inform. Sci., 7:95-132.

Ingo Schröder, Kilian A. Foth, and Michael Schulz. 1999. [X]cdg Benutzerhandbuch. Technical Report Dawai-HH-13, Universität Hamburg.

Ingo Schröder. 2002. *Natural Language Parsing with Graded Constraints*. Ph.D. thesis, Dept. of Computer Science, University of Hamburg, Germany.

David Waltz. 1975. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw–Hill, New York.

Mats Wirén. 1993. Bounded incremental parsing. In *Proc. 6th Twente Workshop on Language Technology*, pages 145–156, Enschede/Netherlands.