Evaluation of the Stochastic Extension of a Constraint-Based Dependency Parser

Renjini Narendranath

Bachelor's Thesis

Dept. of Computational Linguistics
Universität des Saarlandes, Germany

October, 2004

Statement Under Oath

I hereby declare that the work presented in this thesis was completely done by me and that I have not used any sources of help other than those explicitly mentioned.

Renjini Narendranath

October 2, 2004
Saarbrücken

Abstract

In this thesis we experimentally evaluate the performance of a Stochastic extension of the Extensible Dependency Grammar (SXDG) solver, a constrained-based dependency parser. We test the performance of the parser on sentences and grammars automatically acquired from the Penn Treebank and evaluate how the stochastic guidance helps the parser to prune the search tree. Our experimental study reveals that the stochastically enhanced parser (SXDG) is able to prune the search tree considerably, but that the current setup of SXDG still has potential for improvements. We also give directions for future improvements of the current SXDG parser.

Acknowledgments

I am grateful to Prof. Dr. Manfred Pinkal for giving me an opportunity to do my Bachelor's thesis under his supervision. I am heavily indebted to Marco Kuhlmann for guiding me throughout the past three months of my thesis work. I once again thank him for his many lessons, for motivating me to take up this exciting project and for carefully going through my thesis and giving me precious suggestions. I sincerely thank Prof. Dr. Gert Smolka for allowing me to use the excellent resources of the Programming Systems Lab. Finally, I thank my parents, my sister, all my friends and relatives, especially my husband Naveen, for all the love and emotional support.

Contents

1. Introduction 1

2. XDG and SXDG 5

2.1 Topological Dependency Grammar (TDG) 5

2.2 Extensible Dependency Grammar (XDG) 6

2.3 Statistical SXDG Parser 8

2.4 Architecture of the SXDG Parser 8

3. Experimental Evaluation of SXDG Parser 10

3.1 Execution Setup of SXDG Parser 10

3.2 Experimental Setup 12

3.3 Experimental Results 13

4. Discussions 20

4.1 Interpreting the Results 20

4.2 Directions for Future Improvements 24

4.3 Conclusion 25

Bibliography 26

Appendix A: Additional Plots 27

Chapter 1

Introduction

Extensible Dependency Grammar (XDG) [DDK+04a, DDK04b] is a formal framework for dependency grammar which supports the characterization of linguistic information along multiple dimensions of description. In contrast to traditional dependency grammars, XDG is not restricted to specifying grammatical functions. It can also explicitly account for word-order (linear precedence), predicate argument structure, scope structure (for quantifier scope) etc. Parsing with XDG amounts to the selection of a lexicon entry for each word and the configuration of these lexicon entries into a graph or tree, respecting the valencies specified by the lexicon entries, as well as global well-formedness conditions. Using this multi-dimensional paradigm of XDG, we can analyze many complex phenomena like verb extraposition, scrambling etc., in free word-order languages.

The XDG extends the Topological Dependency Grammar (TDG) [DD01] by formulating a novel syntax-semantics interface that maps between the syntax and semantics as a relation and not a function. The current parser implementation for XDG is based on constraint programming [KN02] and supports concurrent flow of information between two levels of linguistic description. This parser has two basic drawbacks. The first drawback is that it performs an exhaustive search in the complete solution space. This is computationally infeasible as the XDG parsing is an NP-complete problem [DDK+04a]. The second drawback is that currently, the XDG parser is used for hand-tailored grammars only. However, for a broader coverage of sentences, we need large grammars acquired from corpora. Handcoding such large grammars is practically infeasible.

To circumvent these drawbacks, a stochastic extension of the XDG parser (the SXDG parser) with the additional capability to automatically acquire a grammar from a corpus was proposed in [DKK03] . The basic idea is to use a probability function, based on the word frequency information learned from a training corpus, and to evaluate choices in search, according to this function. This information is then used by the constraint solver to guide through the search space.

In this thesis, we experimentally evaluate the performance of this proposed extension of XDG parser. We run a number of experiments using sentences and grammars acquired from the Penn Treebank and evaluate different performance parameters of the parser like the parse time, the number of wrong branchings it took, the number of successful branchings etc. Our experimental study reveals that the stochastically enhanced parser is able to prune the search tree considerably, but that the current setup of SXDG still has potential for improvements.

Thesis Outline. Further down in this chapter, we explain some of the basic terminologies used in this thesis. In Chapter 2, we focus on XDG and SXDG and discuss the drawbacks of the current XDG implementation in more detail. We give the architectural overview and the execution setup of the SXDG parser implementation in Chapter 3. Here, we also describe the details of the experiments that we conducted and also provide sample charts of the performance parameters under consideration. Finally, in Chapter 4, we discuss our experimental findings and also provide suggestions for future improvements. Appendix A contains some additional charts that are not presented inside the main chapters.

Basic Terminologies

Dependency Grammar

This thesis is about the evaluation of a processing framework (the SXDG) of dependency grammar. In contrast with phrase structure grammar, where grammatical relations between words (subject, object etc.) emerge from an intermediate constituent structure, in dependency grammar, these relations are expressed directly on the words. Dependency grammar describes asymmetrical dependency relation between words. Figure 1.1 illustrates an example dependency tree. Here, `acquired` is head of the whole sentence and `Simons`, `Georgia` and `fast` are the dependents of the verb as Subject, Object, Adverbial Object respectively.

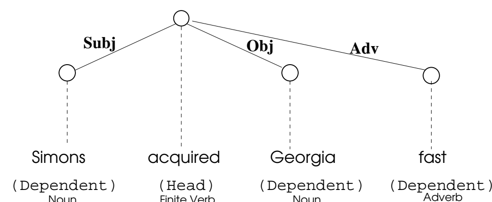


Fig. 1.1: A dependency analysis

Constraint Programming

The basic constraint programming paradigm is to combine existing search methods like backtracking or branch-and-bound with constraint propagation techniques (methods to prune the search space). As the application areas of constraint programming grew in the course of time, new types of constraints were identified and new propagation algorithms for them were developed. For an excellent introduction to the field of constraint programming, we refer to the book by Apt [Apt03].

To define the constraint satisfaction problem formally, we need the following basic definitions. A *variable* X is an object that is associated with a set $Dom(X)$, called the *domain* of X. The domain consists of all values that can be assigned to X. Suppose we have a finite sequence

of variables $\mathcal{X} = [X_1, \ldots, X_n]$ (with $k \geq 1$). (The order of the variables is important and we allow a variable to appear more than once.) A *constraint* C on $\mathcal{X}$ is a tuple $(\mathcal{X}, \mathcal{R})$ such that $\mathcal{R} \subseteq Dom(X_1) \times \ldots \times Dom(X_k)$. We call $\mathcal{X}$ the *variable sequence* of C and denote it by $Vars(C)$, and $\mathcal{R}$ is called the *relation* of C and is denote by $Rel(C)$. We say that the tuple $(d_1, \ldots, d_k)$ *satisfies* C if $(d_1, \ldots, d_k) \in Rel(C)$.

Constraint Satisfaction Problem (CSP). A constraint satisfaction problem (CSP) $\mathcal{P}$ is a tuple $(\mathcal{V}, \mathcal{C})$ such that $\mathcal{V} = \{X_1, \ldots, X_n\}$ is a finite set of variables, $\mathcal{C}$ is a finite set of constraints and each constraint C in $\mathcal{C}$ is a constraint on a sequence of variables in $\mathcal{V}$. We write $\mathcal{P}$ as $\langle X_1 \in Dom(X_1), \ldots, X_n \in Dom(X_n); \mathcal{C}\rangle$. A variable assignment for $\mathcal{P}$ is a mapping $\alpha : \mathcal{V} \rightarrow \cup_{i=1}^{n} Dom(X_i)$ such that $\alpha(X_i) \in Dom(X_i)$ for $i = 1, \ldots, n$. For a sequence $\mathcal{X} = [X_{i_1}, \ldots, X_{i_k}]$ of variables of $\mathcal{V}$, we define $\alpha[\mathcal{X}] := (\alpha(X_{i_1}, \ldots, \alpha(X_{i_k}))$. We say that α satisfies a constraint C in $\mathcal{C}$ if $\alpha[Vars(C)] \in Rel(C)$. α is a solution of $\mathcal{P}$ if it satisfies all constraints in $\mathcal{C}$. We write α as $[X_1 = \alpha(X_1), \ldots, X_n = \alpha(X_n)]$. $\mathcal{P}$ is called consistent if it has a solution and inconsistent otherwise. The search space of $\mathcal{P}$ is defined as the set of all variable assignments for $\mathcal{P}$ and is denoted by $S(\mathcal{P})$.

In many situations, a constraint is not specified as a tuple $((X), \mathcal{R})$ but rather in a symbolic way. For example, "$X < Y$" denotes the constraint $([X, Y], \mathcal{R})$ with $R = \{(x, y) \mid x \in Dom(X) \wedge y \in Dom(Y) \wedge x < y\}$. Thus the interpretation of a symbolic constraint depends on the domains of its variables and on the semantics associated with the constraint.

An example. Consider the following constraint satisfaction problem. We consider the famous puzzle

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

The task is to replace each letter by a different digit such that the summation above becomes correct. In fact, the puzzle has a unique solution as shown below

$$\begin{array}{ccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

This puzzle can be modeled as a CSP as follows. Let S, E, N, D, M, O, R and Y denote the corresponding variables. The domain of S and M are $[1, \ldots, 9]$ and that of the remaining variables is $[0, \ldots, 9]$. The puzzle is formulated as:

$$\begin{array}{rrrrr} & 1000 \cdot S & + 100 \cdot E & + 10 \cdot N & + D \\ + & 1000 \cdot M & + 100 \cdot O & + 10 \cdot R & + E \\ = 10000 \cdot M & + 1000 \cdot O & + 100 \cdot N & + 10 \cdot E & + Y \end{array}$$

This condition can be modeled by the linear equality constraint C_{LE} where $Vars(C_{LE}) = [S, E, N, D, M, O, R, Y]$ and $Rel(C_{LE})$ consists of all tuples $(s, e, n, d, m, o, r, y) \in Dom(S) \times \ldots \times Dom(Y)$ that satisfy

$$9000 \cdot m + 900 \cdot o + 90 \cdot n + y - 1000 \cdot s - 91 \cdot e - 10 \cdot r - d \; = \; 0$$

We still have to express the requirement that the 8 digits are pair-wise different. We could do this with an inequality constraint of the form "$A \neq B$" for each pair of variables, this would yield total of 28 inequality constraint. But we can also use a single constraint which expresses all these constraints at once:

$$C_A = Alldiff(S, E, N, D, M, O, R, Y)$$

We have $Vars(C_A) = Vars(C_{LE})$, and $Rel(C_A)$ consists of all the tuples in $Dom(S) \times \ldots \times Dom(Y)$ where the components are pairwise different.

Two basic operations performed by a constraint solver are *constraint propagation* and *constraint distribution*. The constraint propagation is the process of narrowing the value domain of a variable. When no further propagation is possible then a distribution or *branching* is used. That is, a constraint applied to one branch of the search tree is excluded from the set of constraints applied to the other branch. The constraint chosen for distribution should be such that it facilitates further constraint propagation.

Statistical Natural Language Processing

Due to the inherent difficulty of hand-crafting large sets of rules for processing sentences, researchers creating practical systems that manipulate human languages are turning more frequently to statistical or corpus-based approaches. In statistical natural language processing, the probabilistic informations gained from a corpus is used for decision making while solving the basic disambiguation problems related to word sense, word category, syntactic structure and semantic scope. Increasingly available corpora of natural languages makes such an approach more feasible. Refer [MS01] for instance for a comprehensive treatment of this subject.

Chapter 2

XDG and SXDG

In this chapter, we describe in more detail the XDG grammar formalism and its forerunner Topological Dependency Grammar (TDG). We further discuss about the XDG parser, its drawbacks and the motivation for the SXDG parser. We also present here, the architecture of SXDG parser and give a short description about its modules.

2.1 Topological Dependency Grammar (TDG)

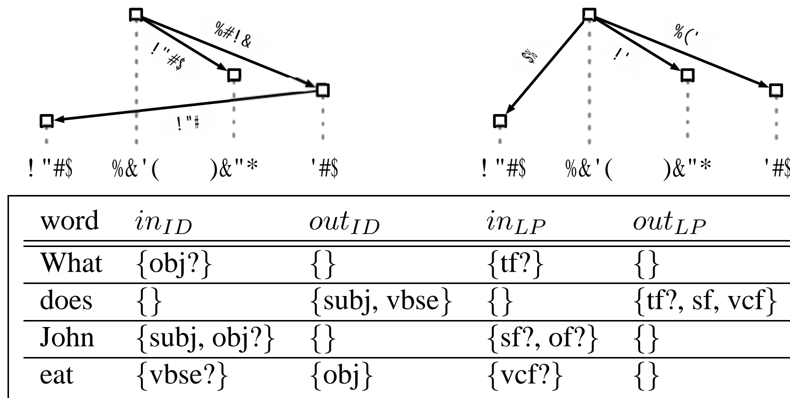
Dependency grammars help in writing simple and elegant grammars for free word-order languages. This is particularly interesting where such languages exhibit complex phenomena of discontinuous constructions, like verb extra position, WH positioning etc. Discontinuity is a phenomenon which causes the constituents of a sentence to scatter in the sentence. This is typical in free word-order languages. The issue of discontinuity is reflected in crossing edges in the dependency trees. For an extensive treatment of discontinuity, we refer the reader to [DD01]. In German (which is a free word-order language) for example, the phenomena of scrambling and extraposition lead to discontinuous VPs (i.e, verb at the end of sentence and its subject somewhere embedded inside the sentence). Phrase Structure Grammar (PSG) handles the issue of crossing edges with the help of traces; and separate rules for each word-order configuration. In Head-Driven Phrase Structure Grammar (HPSG) they are handled with slashed features. Both remedies obscure the dependency structure inherent in the phrase structure analysis.

Topological Dependency Grammar (TDG), can tackle the problem of discontinuity with the help of two orthogonal, yet mutually constraining, structures [DD01]. One of these structure explains the dimension of Immediate Dominance (ID) – this dimension captures the purely syntactic functions of words. The second structure captures the dimension of Linear precedence (LP) – this dimension captures the topological structure of which word precedes which other in a sentence. Both dimensions are related by the principle of climbing where, crossing edges in the ID–tree are allowed to “climb up” and land in a suitable field where in the LP tree they do not cross any edges. The edge labels of the LP–tree are totally ordered in terms of linear precedence. In German, in addition to scrambling and partial extraposition, TDG can handle other linguistic phenomena like the auxiliary flip construction, intermediate placement construction etc. These can be adequately represented with the LP and ID dimension trees.

Formally, the TDG framework consists of a set of lexicalized constraints and principles which govern the “climbing” conditions. Each word in a sentence corresponds to a node. Each node in the ID– and LP–trees has a lexical entry. Nodes are connected with labeled edges. Each lexical entry specifies which valid edge can enter the node and what edges can leave it. The root for instance allows no incoming edges but only allows edges to go out of it.

2.2 Extensible Dependency Grammar (XDG)

The Extensible Dependency Grammar (XDG) framework extends TDG by providing (in addition to the LP– and ID–dimension) multiple dimensions/descriptions like Deep Syntax (DS), Predicate-Argument Structure (PA) and Scope structure (SC) [DDK+04a]. Figure 2.1 describes the XDG analysis of the Sentence “`What does John eat`” along the ID– and LP–dimension. The table in the figure depicts the possible incoming (in_{ID}, in_{LP}) and outgoing (out_{ID}, out_{LP}) edges given a lexical entry . The edges on the ID–dimension corresponds to the syntactic functions (like subject, object, verb base form (vbse)). Here for example, the lexical entry `John` is the subject of the head `does`. The edges on the LP–dimension stand for the topological fields namely, the field for a subject (sf), verb complement field (vcf) and an optional topicalised material (tf).



word	in_{ID}	out_{ID}	in_{LP}	out_{LP}
What	{obj?}	{}	{tf?}	{}
does	{}	{subj, vbse}	{}	{tf?, sf, vcf}
John	{subj, obj?}	{}	{sf?, of?}	{}
eat	{vbse?}	{obj}	{vcf?}	{}

Fig. 2.1: XDG analysis of “`What does John eat`”

An XDG analysis consists of separate graphs for different dimensions, but the same set of nodes. They are constrained by two kinds of parameters: *lexicon* and *principles*. Quoting [DDK+04a] “The *lexicon* of an XDG grammar describes properties local to individual nodes, such as valency. The grammar’s *principles* express constraints global to the graph as a whole such as treeness. Well-formed analyses are graphs that satisfy all constraints.” Therefore, the graphs in figure 2.1 are well formed according to XDG, because they satisfy all the constraints imposed by the grammar (in the table).

As mentioned before, XDG allows graphs along multiple dimensions. The parsing problem of XDG can be stated as a constraint satisfaction problem. The constraint-solver for XDG is

implemented using Mozart/Oz programming system. The solver searches for a solution based on the constraints imposed by the grammar on each node. The parser allows the flow of partial information from one dimension to another, where it can use additional information from one dimension to reduce the ambiguity on the other dimension.

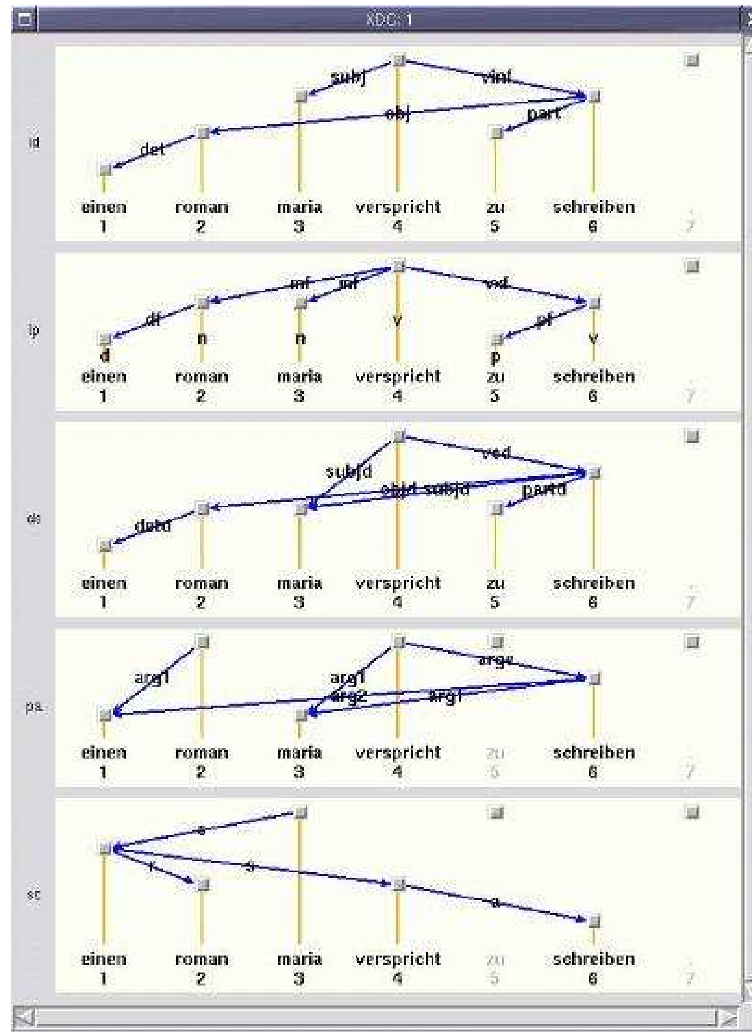


Fig. 2.2: The different dimension graphs

Figure 2.2 illustrates the different dimension graphs produced by the XDG parser while parsing a German sentence:

“`Einen Roman verspricht Maria zu schreiben`”
(A novel promised Maria to write.)
Maria promised to write a novel.

In Figure 2.2, the apart from the ID and LP-dimension, we can see the Deep Structure (DS)

dimension graph. This graph shows the deep syntactic structure, which is a reminiscent of Chomskian deep-structure (D–structure) as in Government and Binding theory (G & B). In G & B, the body of grammar is divided into two blocks: D–structure (the underlying syntactic structure of the lexical entries) and S-Structure (Surface level representation of the sentence). Here, for example, `Maria` is the deep Subject (subjd) of the head `verspricht` (meaning "promised"). Figure 2.2 further depicts the Predicate Argument (PA) Dimension, which specifies the linguistic phenomenon of predicates and its arguments. Here, `schreiben` (meaning "write") has two arguments. The argument-1 is `Maria` and the argument-2 is `einen roman` (meaning "a novel"). The final graph is that of Scope (SC) dimension, which specifies the scope and restriction of lexical entries imposed by Lambda bindings when the sentence is represented in Lambda structures [Deb03].

2.3 Statistical SXDG Parser

Up to now, was only used to parse sentences with handwritten grammars of different sizes, the biggest one being that for German. Handcoding the grammar is a tedious and expensive task. There are corpora, for example the Penn Treebank (PTB), which contains sentences from newspaper texts which are already annotated by hand. The PTB contains the phrase structure trees of sentences along with their Parts-of-Speech (PoS) tags. Treebanks are valuable instruments for linguistic studies. This is a motivation for using the informations from such corpora and extracting grammars for the XDG parser automatically, so that it can handle large-scale grammar and parse sentences, possibly also sentences with words unseen by the grammar. Once the grammar is richer with more words from the corpus, this would help the parser in handling a wider variety of sentences. Because of the largeness of such a grammar, the need for a statistical guidance arises, which can guide the parser without having to search the whole solution space. Because of the heavily lexicalized nature of the XDG parser, it cannot handle the sentences whose words are not specified in the lexicon. In the event of the parser not having seen a particular combination of words, it would make the parser robust if there was some kind of "*Oracle*" which could guide the parser with stochastic informations about solutions in order to perform search. The statistical Extensible Dependency Grammar (SXDG) parser is an extension of the XDG parser that can use a probability function to calculate costs and set choices in search for guiding the constraint solver into the branches with the lowest cost. In Chapter 2.4, we describe the architecture of the SXDG parser, whose performance evaluation is the central theme of this thesis.

2.4 Architecture of the SXDG Parser

In this section, we give an overview of the architecture of the SXDG parser and its modules.

Figure 2.4 depicts the current architecture of the SXDG parser. The SXDG parser is an extension of the XDG parser using a probability model for the ID– and LP–Dimension and a

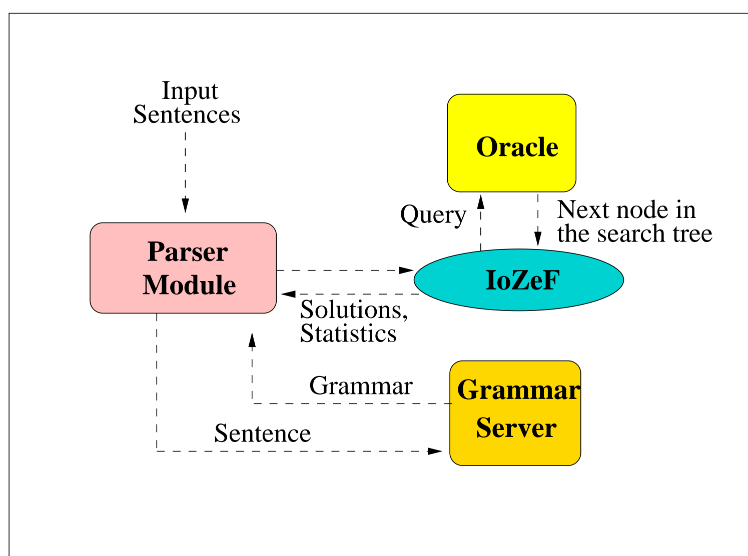


Fig. 2.3: The architecture of SXDG parser

cost estimation function for the partial trees in the search tree. The cost function computes the probabilities of the best next node to continue search, until the best (most probable) complete parse tree of a sentence is reached. The SXDG parser comprises of the following modules; (for more details, refer [DKK03, DDK$^+$04a].

Parser Module. This is the main XDG parser module which parses the input sentence based on the XDG grammar and searches for solutions (complete parses).

The Oracle. This module is responsible for guiding the search using a probability model for dependency trees. That is, it follows an A* search control regime for the parsing algorithm which uses an evaluation function to estimate the costs based on the probability informations for the best next node to continue search in. It maintains an agenda (priority queue) from which it pops the search node with the lowest cost. Such usage of stochastic information is an admissible heuristic for A* search and it helps to infer and cut–short large branches of the search trees, thus reducing the size of the search space.

Grammar Server. For each sentence, the grammar server, from its input data, provides a grammar that contains possible lexical entries for the words in that sentence. This module is also responsible for converting the phrase structures from the Penn Treebank format to XDG format.

IOzSeF Search Module. This module provides a generic interface to search in the constraint programming systems. IOzSeF provides an explorer which gives a visualization of the search tree where we can see how the search has proceed. The IOzSeF queries the Oracle and gives back the solutions to the parser.

Chapter 3

Experimental Evaluation of SXDG Parser

In this chapter, we discuss the execution setup of SXDG parser, employed in our experiments, where we show how the different modules are distributed across machines. We give a short description of the input and output format followed by the parser and the currently inbuilt search strategies. Further in this chapter, we describe the results of the experiments conducted in order to evaluate the performance of the SXDG parser. By these experiments, we test the hypothesis : A statistical guidance would enable the constraint solver of the XDG parser to output the best parse result with improved performance. More precisely, we evaluate the following four performance parameters of SXDG parser.

1. *Time taken for parsing.*
2. *Maximum size of the agenda.*
3. *Total number of solutions* encountered while traversing the search space.
4. *Total number of failures,* which is the number of times the traversal met with an inconsistency with the grammar.

3.1 Execution Setup of SXDG Parser

In order to increase the efficiency of parsing process and to make it less vulnerable to differences in a single machine's workload, the different components involved in the parsing were distributed among multiple machines as shown in Figure 3.1. The SXDG parser was run on one machine and, the Oracle and the Grammar server were run on another machine. The processes residing in different machines communicated with each other via sockets. Both Oracle and the Grammar server ran in a listening mode, i.e., they listen for incoming requests on predefined sockets. Upon receiving a request from SXDG parser, they serve the request and communicate back the results.

Input to SXDG parser

The input data to SXDG parser is a set of sentences extracted from different sections of the Penn Treebank. The input sentences should follow a predefined format, which is different from the

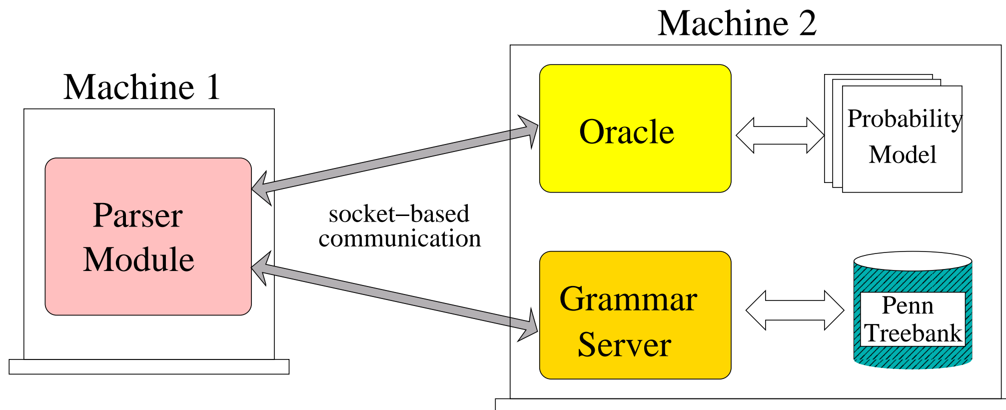


Fig. 3.1: The different components of SXDG parser. The SXDG main parser module in one machine and, the Oracle and the Grammar server run on another machine.

sentence format followed in Penn Treebank. For example, consider the sentence: `Scientists felt differently`. In the Penn Treebank, this sentence is annotated in the following way.

((S
(NP-SBJ (NNS `Scientists`))
(VP (VBD `felt`)
(ADVP-MNR (RB `differently`))) (. .)))

The same sentence is input to SXDG parser is as follows.

`Scientists`_NNS `felt`_VBD `differently`_RB .

Since the phrasal constituents are not needed for the dependency analysis, only the words along with their POS-tags (Parts Of Speech tags) were fed to SXDG parser.

The SXDG Output

For each sentence, the SXDG parser can output its dependency analysis either as an XML record or as a graphical tree representation with separate graph for each dimension (see Figure 2.2 for a sample output). The XML output contains only the dependency analysis for the ID-dimension. For example, Figure 3.2 shows the XML output and the corresponding graphical representation (of the ID-dimension) for the sentence "`Scientists felt differently`".

In addition to the dependency analysis, SXDG parser also outputs for each sentence, various statistics about the parsing process like the time taken for parsing, maximum size of the agenda, the total number of solutions encountered while traversing the search space and the number of times the traversal met with an inconsistency with the grammar.

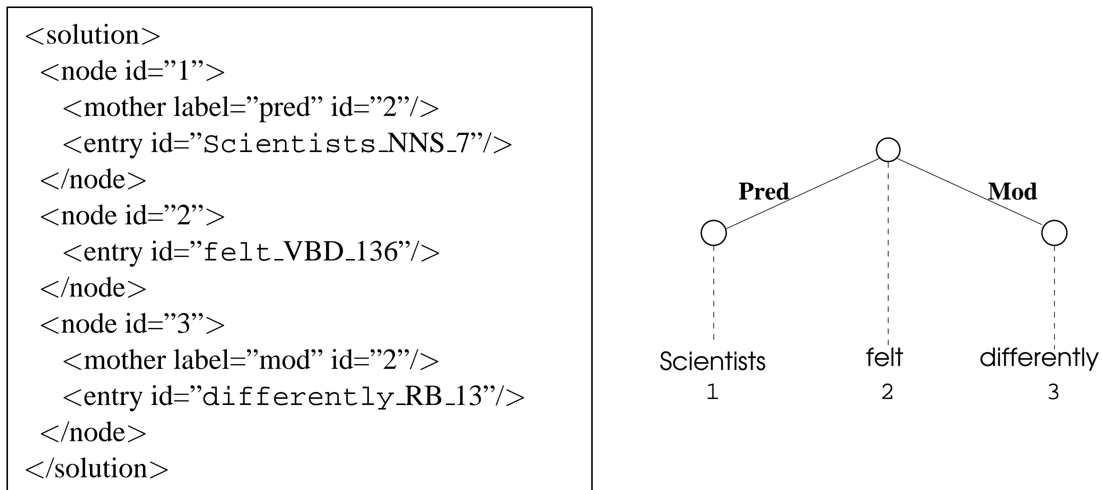


Fig. 3.2: The XML output by SXDG parser and its corresponding graphical representation.

Different Search Strategies of SXDG Parser

The SXDG parser can be configured to run in any one of the following three search modes.

1. *Oracle Based.* In this mode, SXDG parser consults the Oracle to guide its traversal in the search space. The traversal stops when the best solution is encountered.

2. *Depth First.* In this mode, SXDG parser explores the search tree using depth-first in-order traversal. Here the traversal stops when the first solution is reached (this may not be the best solution).

3. *All Solutions.* SXDG parser enumerates all solutions by traversing the complete search space.

3.2 Experimental Setup

An experiment comprise of inputting two sets of sentences to SXDG parser, which are *a)* the “training data” and *b)* the “test data”. The “training data” is used by the parser to extract grammar and also to prepare the probability counts. Using the grammar and probability counts thus obtained, the parser parses the “test data” and output its dependency analysis. Refer Section 3.1 for details on the input and output specifications of SXDG parser.

For all our experiments, we used sentences extracted from the Wall Street Journal (WSJ) sections of the Penn Treebank as “test data” and “training data”. More specifically, we used sentences of length 3–7 from sections 2–22 of the WSJ. We classified the sections 2-10 of WSJ as the the “*training data*” and always used it to prepare the Oracle and the Grammar server before running the experiments. Hence in such a setup, sections 2–10 of WSJ are “known” and sections 11–22 are “unknown” (unknown to the probability model and grammar) to the parser. Thus we

distinguish the performance of the parser on the sections 11-22 of WSJ from that on sections 2-10 of WSJ.

The machine where SXDG parser was run, had the following configuration: Linux O.S with 2.4.22-1.2140 kernel, 700 MHz Athelon processor and 384 MB RAM. The machine where the Oracle and Grammar server were run, had the following configuration: Linux O.S with 2.4.20-27.9smp kernel, 1 GB RAM, Pentium III dual processor, 1 GHz each.

The amount of memory that the Oracle can use could be controlled externally. In all our experiments, the maximum memory allowed for the Oracle was 256 MB. We also specified a timeout of 6 minutes for SXDG parser to parse a sentence. This is to ensure that we are able to run our experiments on a large number of sentences in a reasonable amount of time. If it takes more than 6 minutes, the sentence parsing will be aborted and SXDG parser moves on to the next sentence.

3.3 Experimental Results

Our experimental results speak about the four performance parameters that we discussed in the beginning of this chapter. We tabulate the performance statistics in the form of following two kinds of graphs.

a) Plot of the average performance averaged over the candidate sentences.

b) Plot of the percentage of candidate sentences where the parser timed out (SXDG parser took more than 6 minutes to prase them).

We generated the above statistics for different candidate collections chosen carefully, whose details are discussed next.

Based on the candidate sentences and also based on the settings of SXDG parser that we specified, namely the search strategy and the LP-dimension enable/disable status, the experiments that we conducted can be classified into seven different categories. We denote them as categories A to G and we discuss them below.

We recall that in all the experiments, the input sentences were drawn from the Wall Street Journal(WSJ), sections 2–22, of the Penn Treebank. Moreover, in all the experiments where SXDG parser was configured to use the Oracle strategy, the Oracle was running on the probability information retrieved from the WSJ sections 2–10 of the Penn Treebank.

Category A) Input sections 2–10, lengths 3–7, with Oracle, and LP-dimension ON.

In this round of experiments, SXDG parser was configured to parse the sentences with the LP-dimension switched on. Sentences of lengths 3–7 from section 2–10 of WSJ were issued to SXDG. The search strategy was Oracle based. The figure 3.3 shows the plot of the performance parameters, time taken and the total no. of failures. We refer the reader to the Appendix for the

plots on the remaining parameters. The figure shows the average values, averaged over all sentences of a fixed length. The upper and the lower envelops shown in the plots are the boundaries with respect to the standard deviation values.

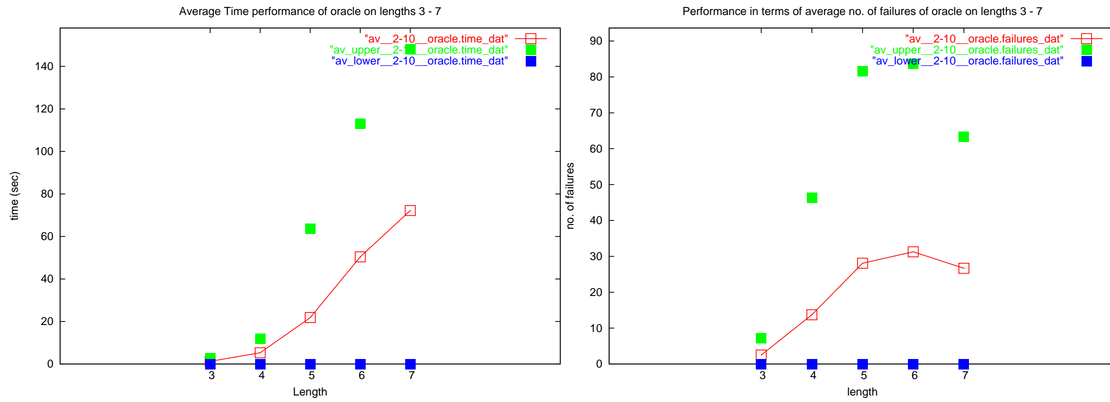


Fig. 3.3: Category A, plot of the time taken and no. of failures.

The sentences for which SXDG parser timed out were omitted from the average value calculation. For such sentences, we generate a separate plot, refer Figure 3.4, showing the percentage of such sentences for each length.

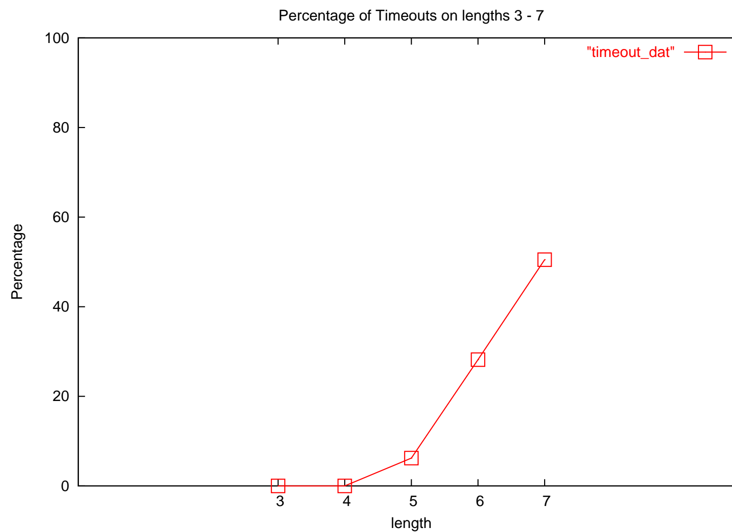


Fig. 3.4: The percentage of sentences timed out.

Remarks about the plots. In all the experiments, the average plots (plot of average values) are generated after omitting those sentences for which the Oracle timed out. The timed out sentences are separately captured in a separate time out plot. In the average plots, the values are obtained by averaging over all sentences of fixed length. In addition to the average values, we plot the

upper and the lower envelops which denotes the boundaries with respect to the standard deviation values. For all the plots that are not shown in this chapter, we refer the reader to the Appendix.

Note that in category A, the input sentences are already known to the Oracle and the Grammar server. To study the performance of SXDG parser on sentences unseen by the grammar, we issued sentences from sections 11-22 of the WSJ, which is the next category.

Category B) Input sections 11–22, with Oracle and LP-dimension ON.

Here also the LP-dimension was switched on. Sentences of lengths 3–7 from section 11–22 of WSJ were issued to SXDG parser. The search strategy was Oracle based. The figure 3.5 shows the plot of the performance parameters, time taken and the total no. of failures.

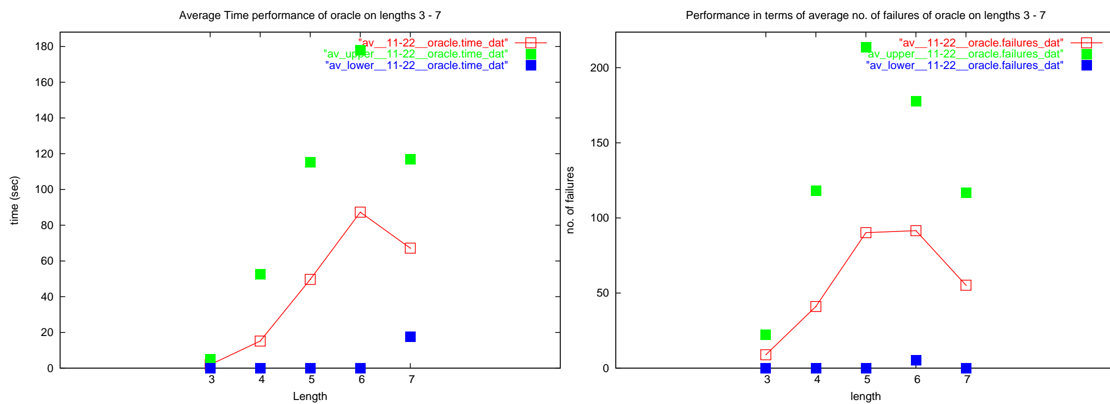


Fig. 3.5: Category B, Plot of the time taken and no. of failures.

Our next objective was to investigate the effect of LP-dimension on/off status on the performance of SXDG parser. To this end, we repeated the above two sets of experiments with LP-dimension switched off; they are categories C and D.

Category C) Input sections 2–10, with Oracle and LP-dimension OFF.

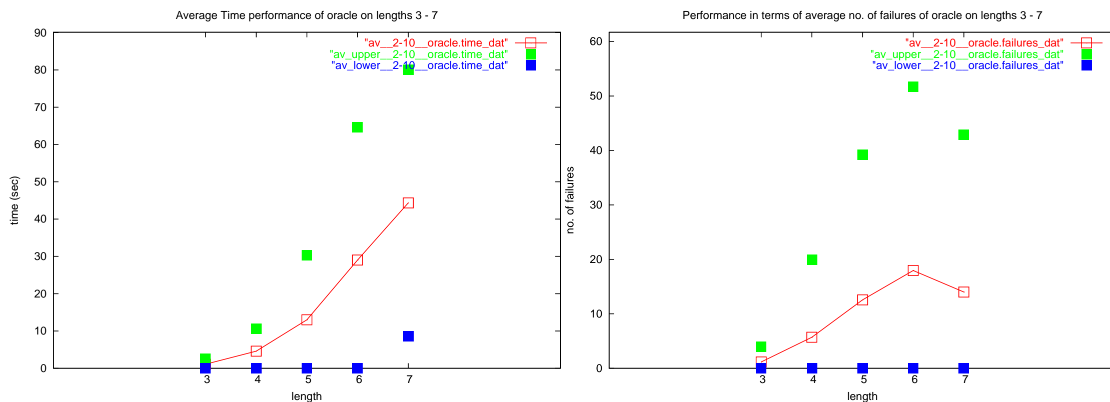


Fig. 3.6: Category C, Plot of the time taken and no. of failures.

The SXDG parser was configured to parse the sentences with The LP-dimension switched off. By this, we wanted to investigate whether this change has any influence on the any of the performance parameters. Sentences of lengths 3–7 from section 2-10 of WSJ were issued. The search strategy was Oracle based. The figure 3.6 shows the plot of the performance parameters, time taken and the total no. of failures.

Category D) Input sections 11–22, with Oracle and LP-dimension OFF.

Sentences of lengths 3–7 from section 11-22 of WSJ were issued to the parser. The search strategy was Oracle based. The figure 3.7 shows the plot of the performance parameters, time taken and the total no. of failures.

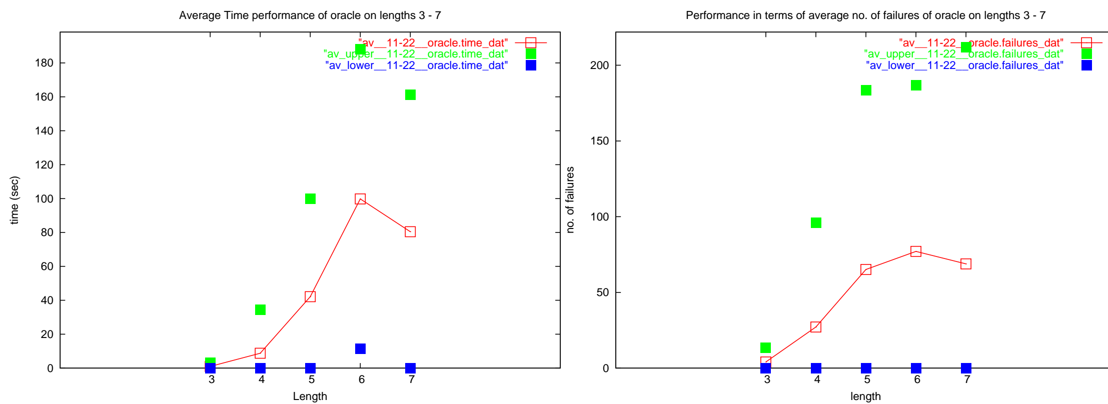


Fig. 3.7: Category D, Plot of the time taken and no. of failures.

As a next step, for a comparison against the depth-first search strategy we performed the following set of experiments.

Category E) Input sections 2–10, with Depth-first and LP-dimension OFF.

Here also SXDG parser was configured to parse the sentences with the LP-dimension switched off. Sentences of lengths 3–7 from section 11-22 of WSJ were issued. The main difference from those above is that the search strategy was depth-first based. The depth-first based search gives the first solution as it traverses the search space in the depth-first fashion. The depth-first strategy never uses the informations from Oracle during search. The figure 3.8 shows the plot of the performance parameters, time taken and the total no. of failures.

One of the questions that we were interested in is, “how much portion of the search tree is cut out by the Oracle based search?”. To answer this question, it makes sense to compare the performance of Oracle based search to the all solutions based search strategy. For this we conducted the following two sets of experiments.

Category F) Input sections 2–10, with All solutions mode and LP-dimension ON.

Sentences of lengths 3–7 from section 2–10 of WSJ were issued to the parser with LP-dimension switched on. The search strategy was all solutions based. In this mode, SXDG parser

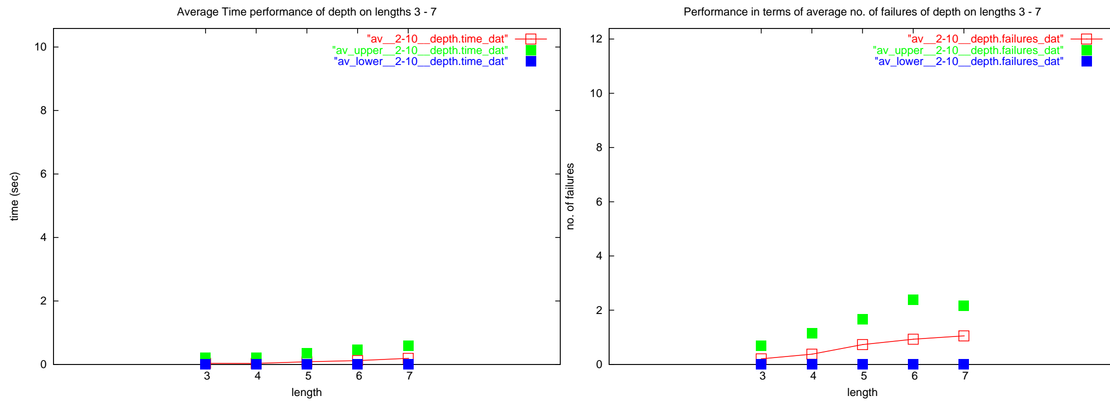


Fig. 3.8: Category E, plot of the time taken and no. of failures.

enumerates all solutions by traversing the complete search space. The figure 3.9 shows the plot of the performance parameters, time taken and the total no. of failures. All input sentences of length 6 and 7 were timed out. Hence they are not shown in the plots below.

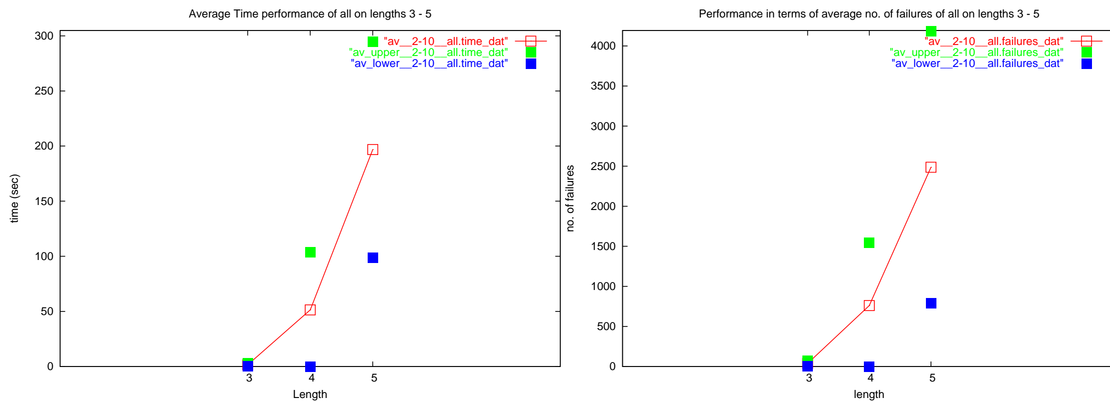


Fig. 3.9: Category F, plot of the time taken and no. of failures.

Category G) Input sections 11–22, with All solutions mode and LP-dimension ON.

Sentences of lengths 3–7 from section 11–22 of WSJ were issued to the parser. Here also the search strategy was all solutions based. The figure 3.10 shows the plot of the performance parameters, time taken and the total no. of failures. Here also, all input sentences of length 6 and 7 were timed out. Hence they are not shown in the plots below.

An Enhanced Probability Model

In case of unseen word configurations (which is found by the constraint solver during constraint distribution), the Oracle assigns a very high cost to it. Instead, it was proposed that these costs for unseen entries can be calculated as a function of the number of lexical entries known to the Oracle, so that when the number of known lexical entries are less, the unseen word-configuration

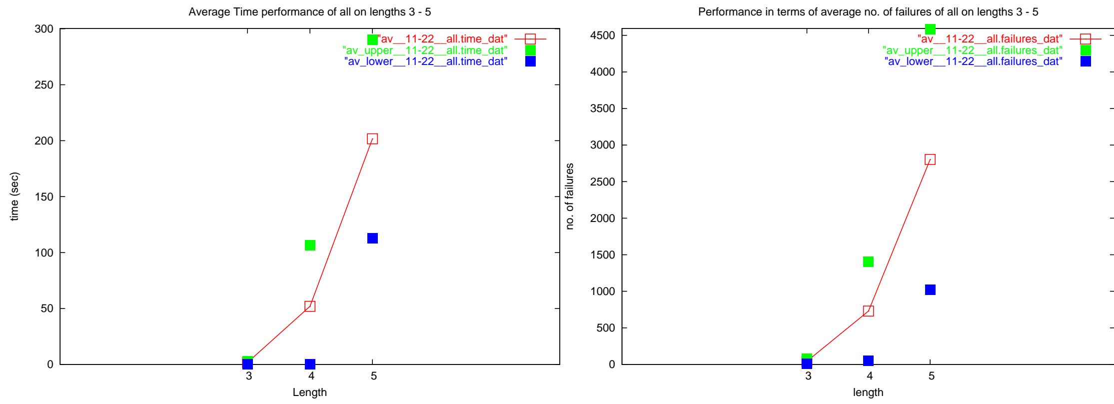


Fig. 3.10: Category G, plot of the time taken and no. of failures.

is not penalized heavily. We denote this enhancement by Oracle+. We performed the below two sets of experiments to test this enhancement.

Category H) Input sections 2–10, with Oracle+ and LP-dimension ON.

The SXDG parser was configured to parse the sentences with the LP-dimension switched on. By this, we wanted to investigate whether this change has any influence on the any of the performance parameters. Sentences of lengths 3–7 from section 2-10 of WSJ were issued. The search strategy was Oracle based. The figure 3.11 shows the plot of the performance parameters, time taken and the total no. of failures.

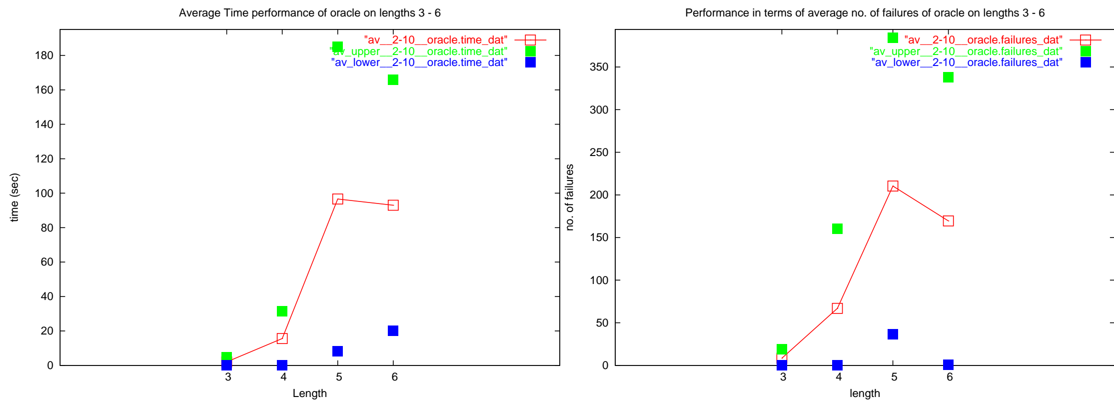


Fig. 3.11: Category H, Plot of the time taken and no. of failures.

Category I) Input sections 11–22, with Oracle+ and LP-dimension ON.

Here SXDG parser was configured to parse the sentences with the LP-dimension switched on. Sentences of lengths 3–7 from section 11-22 of WSJ were issued. The search strategy was Oracle based. The figure 3.12 shows the plot of the performance parameters, time taken and the total no. of failures.

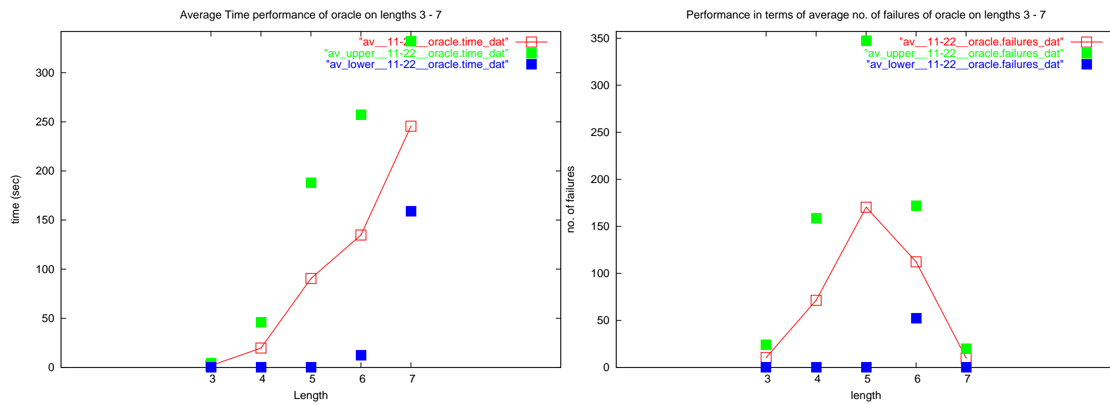


Fig. 3.12: Category I, Plot of the time taken and no. of failures.

In a nutshell, our experiments can be classified as shown in the table below.

Category	Input Sections	Sentence Lengths	Training Data	Grammar Server	Search Mode	LP-dimension ON/OFF
A	2–10	3–7	2–10	2–10	Oracle	ON
B	11–22	3–7	2–10	2–10	Oracle	ON
C	2–10	3–7	2–10	2–10	Oracle	OFF
D	11–22	3–7	2–10	2–10	Oracle	OFF
E	2–10	3–7	2–10	2–10	Depth-first	OFF
F	2–10	3–7	2–10	2–10	All Solutions	ON
G	11–22	3–7	2–10	2–10	All Solutions	ON
H	2–10	3–7	2–10	2–10	Oracle+	ON
I	11–22	3–7	2–10	2–10	Oracle+	ON

Chapter 4

Discussions

The main intuition behind extending the XDG parser was that, an Oracle based on stochastic information could guide the parser while traversing the search space, and cut short the subtrees so as to reduce parsing time and memory consumption. Earlier, the XDG used to be input with hand-crafted grammar in the XDG format. Using the SXDG parser, we could make use of large grammars induced from the Penn Treebank's Wall Street Journal Corpus. In the following, we interpret the experimental results that we obtained.

4.1 Interpreting the Results

General Remark about the Graphs. In some of the plots that have been generated, one can observe that the curve goes down after some point as the sentence length increases. This is contrary to what is expected. We remark that this does not convey the true behavior of the performance parameter. In such situations, a very large percentage of sentences of large lengths were timed out. Only those which are not timed out take part in the calculation of the relevant parameters. We direct the reader to take into account the timeout charts (in the Appendix A) also while reading such graphs.

Categories A and B *(Input sections 2–10 and 11–22, LP-dim ON).* Our objective for running categories A and B was to see how the parser performs on unseen words/word-configurations when compared to its performance on known words/word-configurations. In category A, the training data (sections 2–10) itself was used as the input sentences, whereas in category B the input sentences (sections 11–22) were different from the training data.

As is expected, when the sentence length increase, the time taken for the parsing and the other performance parameters increase as well. The plots (Figure 3.3 and 3.5) suggests that the parse time is super linear in the sentence lengths. That is, the rate of growth in the parse time does not seem to be a constant. The same behavior also holds for the plots on percentage of timeouts (Figures 3.4 and 4.13). There is no time out for sentences of length three, whereas already for sentences of length seven, 50% of the sentences timed out.

The parser takes more time on unseen sentences (Category B), than in the case where the input is same as the training data (Category A). The reason for this could be the following. The

Oracle assigns very high cost for unseen word configurations introduced by the constraint solver. Because of this, the parser might be highly biased against unseen words, which forces the parser to abandon good subtrees and traverse more in the search space to find the best parse, resulting in an increase in parse time. The plots (Figures 3.4 and 4.13) show that for sentences of length seven, in category A, the percentage time out is around 50%. But in category B, the percentage timeout on length seven sentences is close to 90%. The same order of difference also holds for sentences of lengths 3–6.

The number of failures seems to be almost linear in Categories A and B (Figures 3.3 and 3.5). But in Category B, (i.e., on unseen sentences), the number of failures increase by a factor close to three compared to category A. Figure 4.1 compares the time as well as the no. of failures performance for Categories A and B.

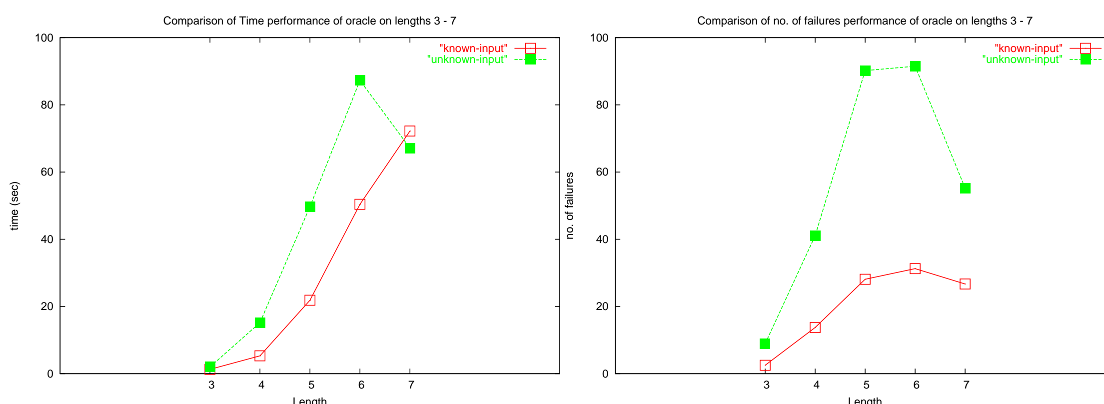


Fig. 4.1: Comparison of the parse time and the no. of failures performance between Categories A (known-input curve) and B (unknown-input curve) with the LP-dimension on.

The parameters, maximum agenda size and the no. of solutions also show a linear growth in terms of the sentence length in both Categories A and B (Figures 4.4 and 4.5). But in category B, it is only slightly worse than Category A.

To summarize, the experiments suggests that difference in performance of the SXDG parser on unseen sentences and seen sentences (training data) is mainly reflected on the parse time and the number of failures.

Our next goal was to investigate the effect of LP-dimension on the parser performance. For this, we performed experiments similar to categories A and B, except that the LP-dimension was switched off. The parser ran on the ID-dimension only. We summarize our findings in categories C and D.

Categories C and D *(Input sections 2–10 and 11–22, LP-dim OFF).* The main difference from the above discussed categories is that in these two categories is that, in categories C and D, the LP-dimension was switched off. That means, only the ID-dimension was enabled. Comparing the time performance of Category C to Category A (Figures 3.6 and 3.3) and Category D to Category B (Figures 3.7 and 3.5), and also taking into account the time outs (Figures 3.4, 4.13,

4.14 and 4.15), it can be deducted that, when the LP-dimension is switch off, the parse time improves slightly.

The curve for the number of failures (Figures 3.3 and 3.6, and Figures 3.5 and 3.7) is almost similar to the situation where LP-dimension was switched on. The values are slightly smaller in the case of LP-dimension switch off.

For maxsize and solutions, the effect seems to be in the opposite direction. That is, when the LP-dimension is switched off, the maximum agenda size and the number of solutions are more than the case when LP-dimension is switched ON. This is true for case where the input is the training data as well as the case where the input is unseen data (Figures 4.4 and 4.6, and Figures 4.5 and 4.7). A possible reason is that the flow of information from one dimension, i.e., LP-dimension, is unavailable to the other dimension, i.e., ID-dimension. This makes the constraint solver to work with smaller number of propagators. This could result in an increase agenda size.

To evaluate the influence of switching off the LP-dimension, Figure 4.2 shows the comparison of time performance between categories A and C as well as categories B and D. The commonality in A and C is that in both cases the input is known. Similarly for both categories B and D, the input is unseen by the grammar.

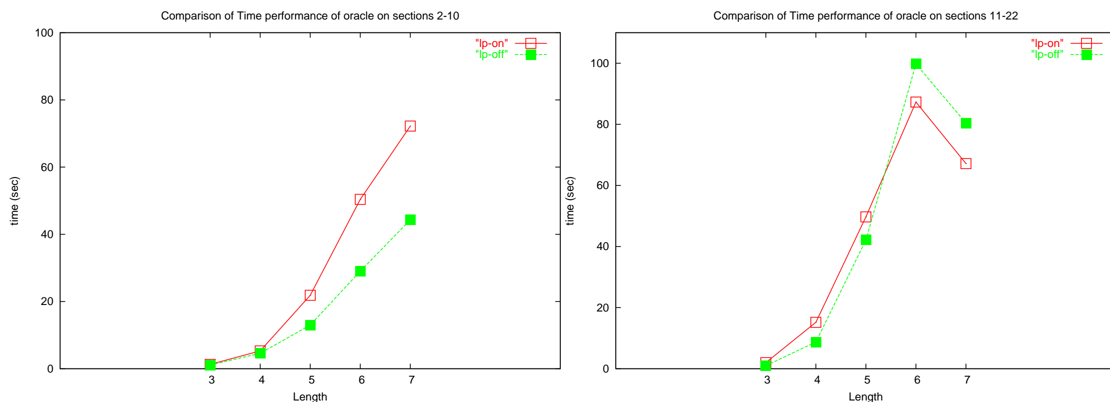


Fig. 4.2: Comparison of the parse time between categories A (lp-on) and C (lp-off) (left fi gure) and between categories B (lp-on) and D (lp-off) (right fi gure).

To summarize, switching off the LP-dimension has positive influence on the parse time and no. of failures, where as it has negative influence on the maximum agenda size and the no. of solutions.

Category E *(Input sections 2–10, Depth-fi rst strategy, LP OFF).* As expected, the time taken to find any solution is much shorter than the time taken to find the best solution. The Figure 3.8 suggest that it needs much more traversal to find the best solution in comparison to the first solution.

Categories F and G *(Sections 2–10 and 11–22, All solutions strategy, LP-dim ON).* Here, our objective was to study the performance of SXDG parser with Oracle against an exhaustive search strategy (all solutions strategy).

It is obvious that if we have to search for all solutions, all the performance parameters will be affected in a negative way. But our goal was to evaluate how bad can the performance parameters become for such a strategy. Such a study reveals the significance of using informed search in the search space in comparison to exhaustive search.

Comparing Category F to Oracle based search on the same input sections, the time taken by the all solutions strategy increases by a factor of around 15 (refer Figures 3.9, 4.17, 3.3, 3.6), the no. of failures increases approximately by factor 100 (Figures 3.9, 3.3 and 3.6) and the no. of solutions increases by factor 1000 (Figures 4.9, 4.4, 4.6). In the similar fashion, if we compare Category G to Oracle based search on the same input sections, the time taken by the all solutions strategy increases by a factor of around 5 (refer Figures 3.10, 4.18, 3.5, 3.7), the no. of failures increases approximately by factor 50 (Figures 3.10, 3.5 and 3.7) and the no. of solutions increases by factor 1000 (Figures 4.10, 4.5, 4.7).

Figure 4.3 shows the comparison of time performance between categories A and F and between categories B and G. In categories A and B, the search strategy is Oracle based. Categories A and F uses the same input sentences. Similarly categories B and G uses the same input sentences.

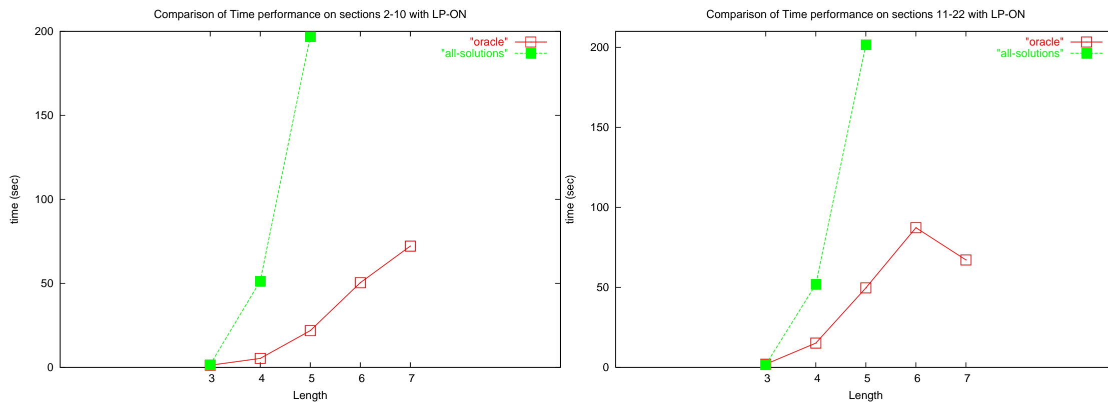


Fig. 4.3: Comparison of the parse time between categories A and F (left fi gure) and between categories B and G (right fi gure).

To summarize, our experimental findings show that our A* search is able to prune the search tree considerably in comparison to exhaustive search where the additional statistical information is not used. This contrast is much more significant when the input data is already known to SXDG parser (that is, input data and training data for Oracle are the same). Thus it supports the argument that the parser enhanced with an Oracle performs much better than parser based on exhaustive search and, training the Oracle with more sentences further improves its performance significantly.

Categories H and I *(Input sections 2–10 and 11–22, Oracle+, LP-dim ON).* In these two categories, we used the enhanced Oracle, i.e., Oracle+ where the unseen words are assigned costs depending on the set of seen words, in order to compare its performance with the earlier Oracle. We compare the charts in Category H to the corresponding ones in Category C, as the only difference between these two setup is that the Oracle is the enhanced one in the Category H. Similarly we compare Category I against Category D.

Upon comparing Categories H and C (in both cases, the input sections are same as the training data), we observe that the Oracle+ performs worse than usual Oracle with respect to all the performance parameters, i.e., parse time, no. of failure, no. of solutions and maxsize. Refer Figures 3.6, 4.6, 4.14, 3.11, 4.11 and 4.19.

Similarly, when we compare Category I to Category D (in both cases, the input sentences are unseen by the Oracle), Oracle+ performed worse than usual Oracle except on the no. of solutions parameter. The number of solutions encountered by Oracle+ is always slightly lesser than the number of solutions encountered by the usual Oracle.

To summarize, enhancing the Oracle to assign the cost to unseen words depending on already seen words seems to have a slight positive impact on the number of solutions seen, but it affects all other parameters negatively. The negative effect is more strong on the parse time.

4.2 Directions for Future Improvements

As of now the SXDG parser cannot handle word-configurations unseen by the grammar. One reason for this could be that, the current constraint solver of the parser controls the distribution. This happens when the constraint solver imposes a constrain φ in the left branch of the search tree during distribution and $\neg\varphi$ in the right branch. This causes the solver to possibly come up with unseen word configurations, which cause the oracle to assign very high cost for it. Because of this, the parser is forced to take bad branchings. That is why we argue that, the oracle should control the distribution. That is, when it encounters an unseen word-configuration, it can assign better values to unseen word-configurations, and thereby avoid bad branchings. Another way to get around this problem is to use smoothing techniques, also known as discounting, where in case of unseen events, we give little bit of probability of seeing it by decreasing the probability of previously seen events. One way of smoothing in our context is, instead of considering the unseen word as such, considering the PoS tag of the word. The chances of PoS tags to have occurred will be considerably much more than the word itself. Another problem is that of Sparseness of data, due to the heavily lexicalized nature of the grammar. Smoothing techniques like considering the PoS tags of the words can be employed to deal with this problem in a better way.

4.3 Conclusion

In this thesis, we evaluated the performance of the statistically enhanced XDG parser using a number of experiments involving sentences from the Penn Treebank. We used a number of performance parameters for this evaluation. As the performance parameters for this evaluation, we used: time taken, the maximum size of the agenda, the number of failed nodes and number of solutions encountered. Our findings suggest that the statistically enhanced parser, which uses an Oracle, is able to prune large portions of the search tree and thus is significantly superior to the exhaustive search approach, which enumerates all the solution. This improvement is more significant when the input words and word configurations are already known to the Parser. We believe that techniques like smoothing (or statistical discounting) and also modifying the constraint solver by involving Oracle also in constraint distribution could further improve the performance of the parser while handling unseen words and word configurations.

Bibliography

[Apt03] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[DD01] Denys Duchier and Ralph Debusmann. Topological dependency trees: A constraint-based account of linear precedence. In *Proceedings of ACL 2001*, Toulouse, France, 9–11July 2001.

[DDK$^+$04a] Ralph Debusmann, Denys Duchier, Alexander Koller, Marco Kuhlmann, Gert Smolka, and Stefan Thater. A relational syntax-semantics interface based on dependency grammar. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING 2004)*, Geneva, Switzerland, 2004. to appear.

[DDK04b] Ralph Debusmann, Denys Duchier, and Geert-Jan M. Kruijff. Extensible dependency grammar: A new methodology. In *Proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar*, 2004.

[Deb03] Ralph Debusmann. Dependency grammar as graph description. In Denys Duchier, editor, *Prospects and Advances of the Syntax/Semantics Interface*, pages 79–84, Nancy, 2003.

[DKK03] Peter Dienes, Alexander Koller, and Marco Kuhlmann. Statistical a-star dependency parsing. In Denys Duchier, editor, *Prospects and Advances of the Syntax/Semantics Interface*, pages 85–89, Nancy, 2003.

[KN02] Alexander Koller and Joachim Niehren. Constraint programming in computational linguistics. In Dave Barker-Plummer, David I. Beaver, Johan van Benthem, and Patrick Scotto di Luzio, editors, *Words, Proofs, and Dialog*, volume 141 of *CSLI Lecture Notes*, pages 95–122. CSLI Press, 2002.

[MS01] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 2001.

Appendix A: Additional Plots

Summary of all the experiment categories.

Category	Input Sections	Sentence Lengths	Training Data	Grammar Server	Search Mode	LP-dimension ON/OFF
A	2–10	3–7	2–10	2–10	Oracle	ON
B	11–22	3–7	2–10	2–10	Oracle	ON
C	2–10	3–7	2–10	2–10	Oracle	OFF
D	11–22	3–7	2–10	2–10	Oracle	OFF
E	2–10	3–7	2–10	2–10	Depth-first	OFF
F	2–10	3–7	2–10	2–10	All Solutions	ON
G	11–22	3–7	2–10	2–10	All Solutions	ON
H	2–10	3–7	2–10	2–10	Oracle+	ON
I	11–22	3–7	2–10	2–10	Oracle+	ON

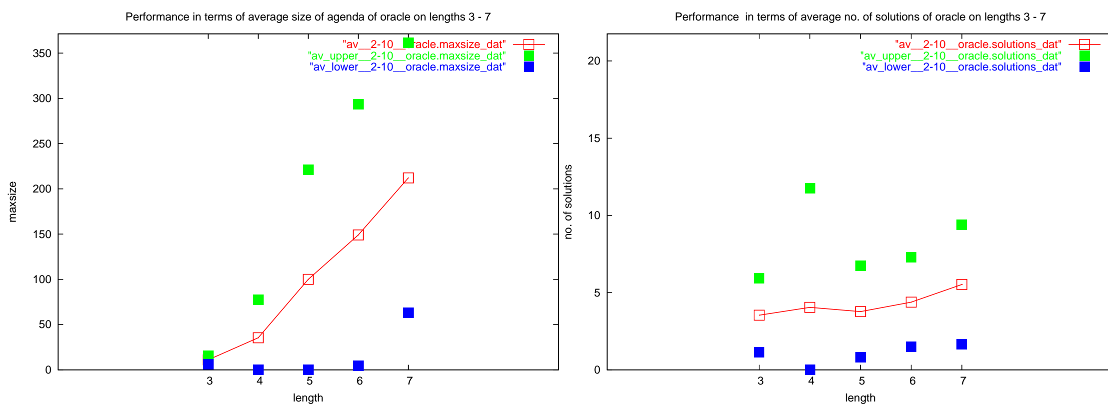


Fig. 4.4: Category A, plot of maxsize and no. of solutions.

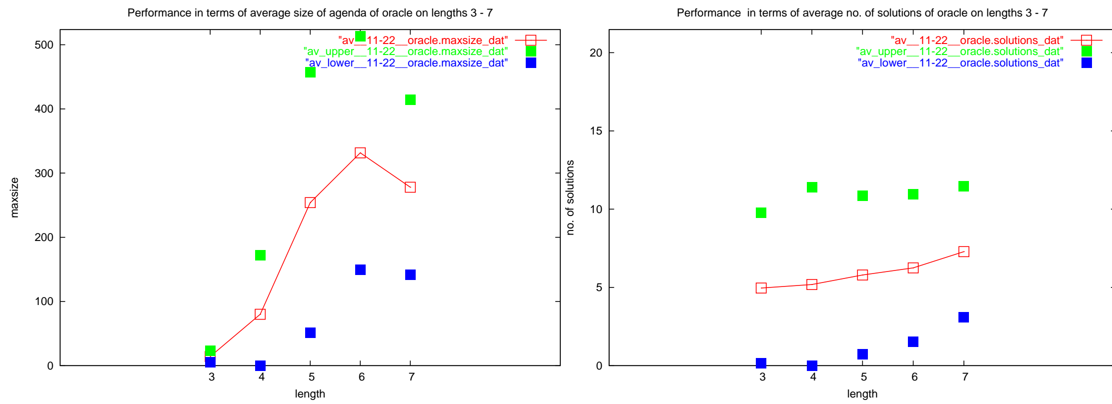


Fig. 4.5: Category B, plot of maxsize and no. of solutions.

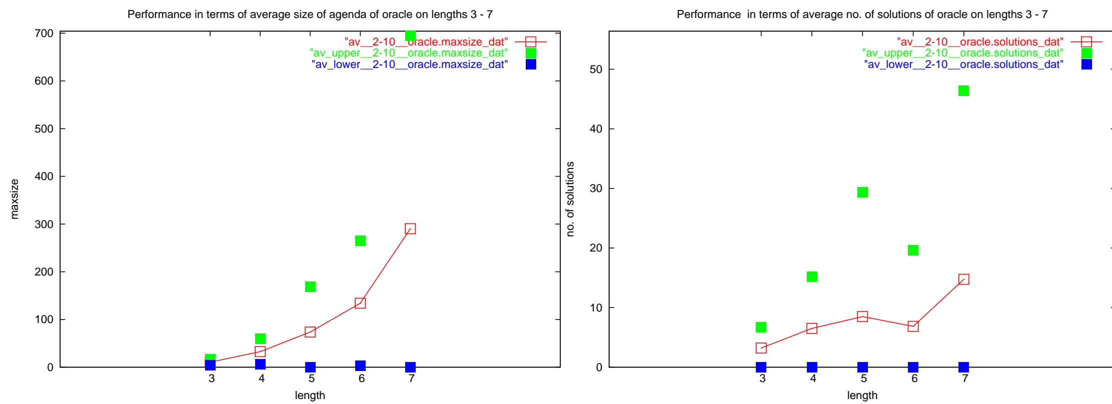


Fig. 4.6: Category C, plot of maxsize and no. of solutions.

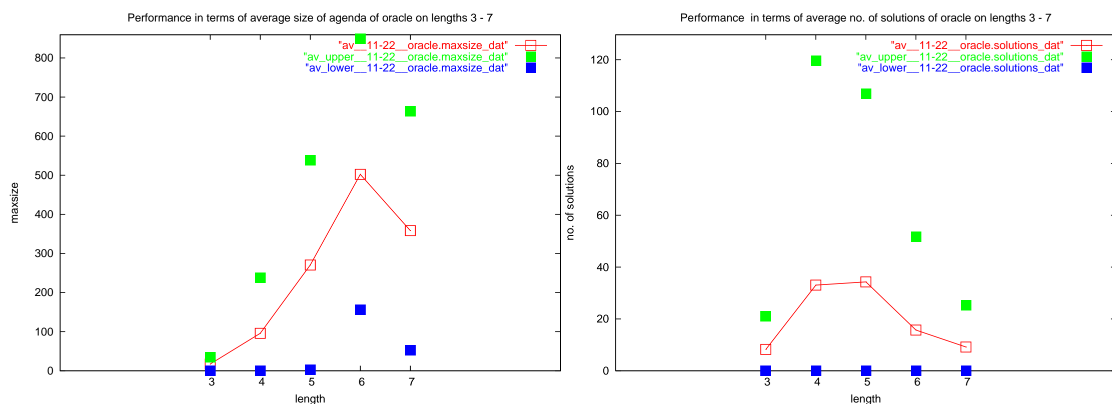


Fig. 4.7: Category D, plot of maxsize and no. of solutions.

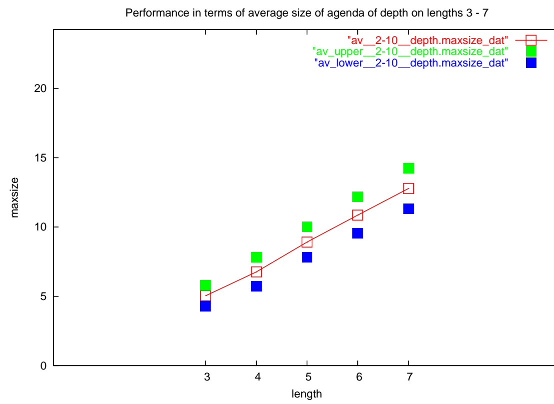


Fig. 4.8: Category E, plot of maxsize.

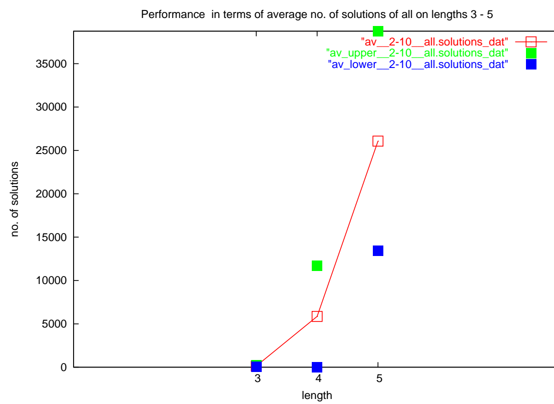


Fig. 4.9: Category F, no. of solutions.

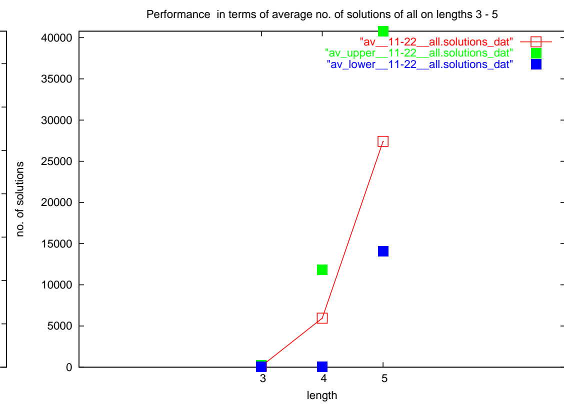
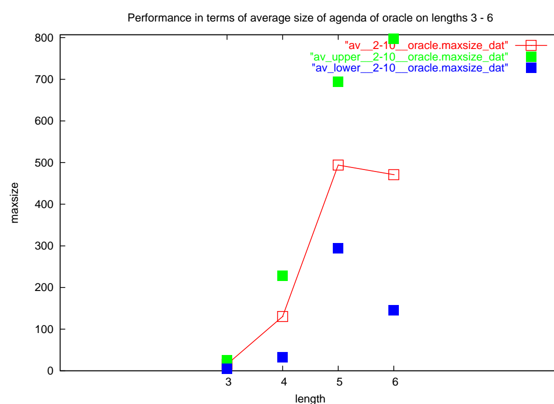


Fig. 4.10: Category G, no. of solutions.



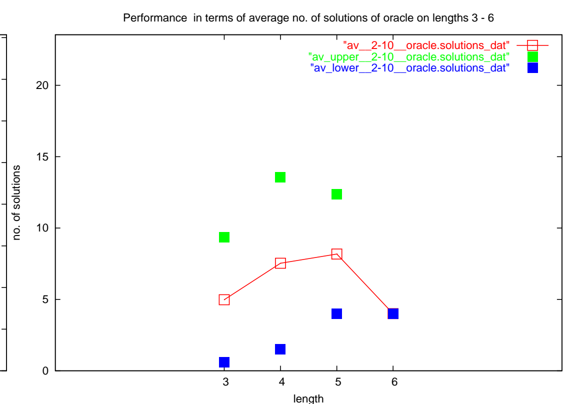


Fig. 4.11: Category H, plot of maxsize and no. of solutions.

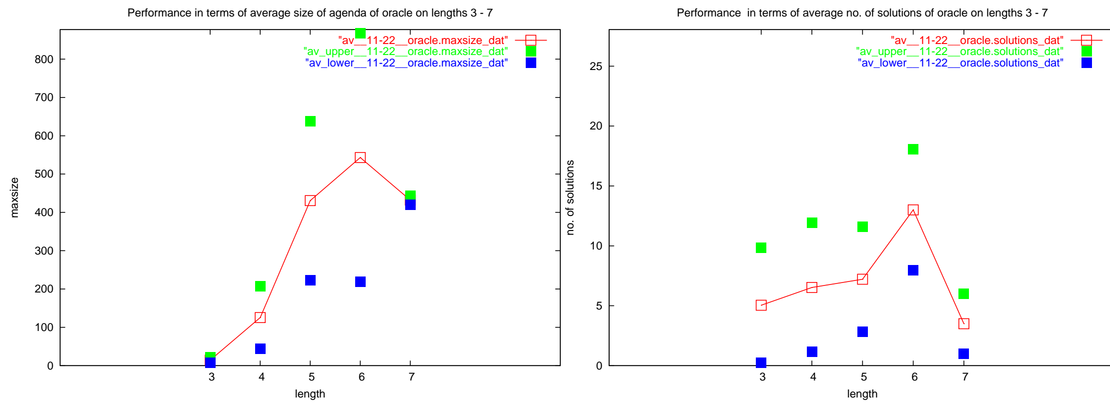


Fig. 4.12: Category I, plot of maxsize and no. of solutions.

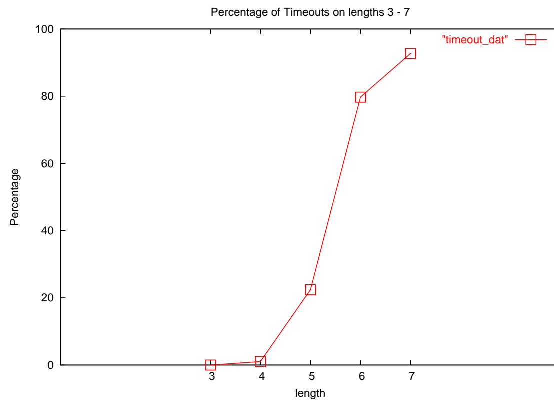


Fig. 4.13: Category B, Timeouts.

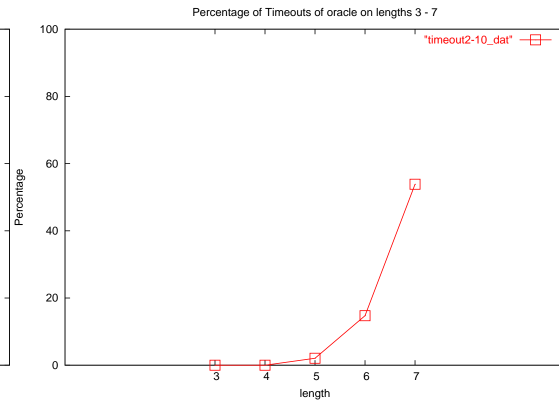


Fig. 4.14: Category C, Timeouts.

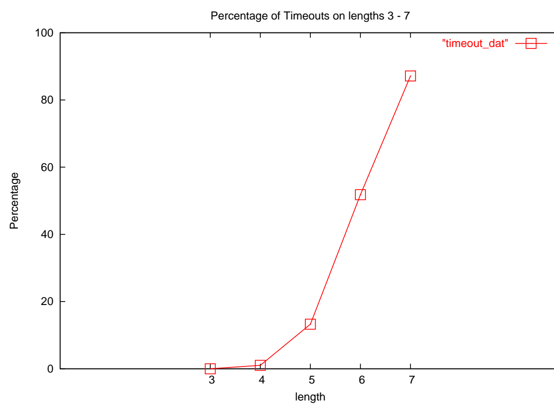


Fig. 4.15: Category D, Timeouts.

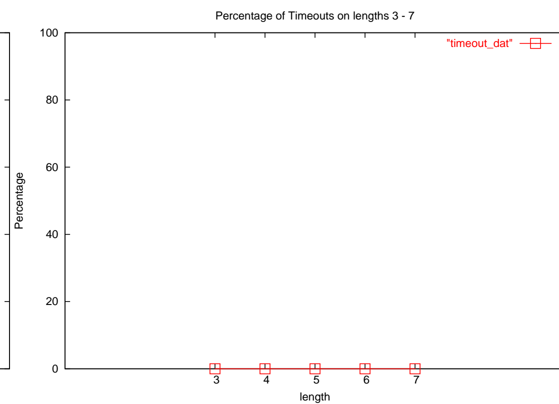


Fig. 4.16: Category E, Timeouts.

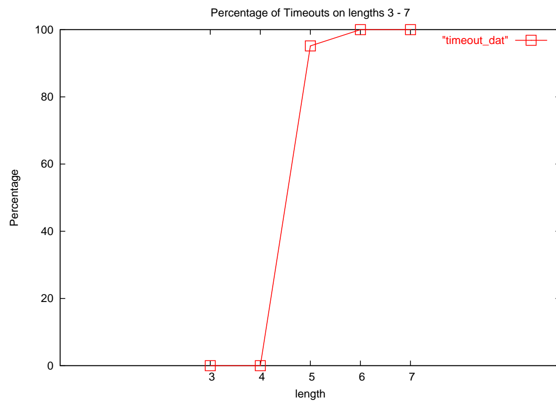


Fig. 4.17: Category F, Timeouts.

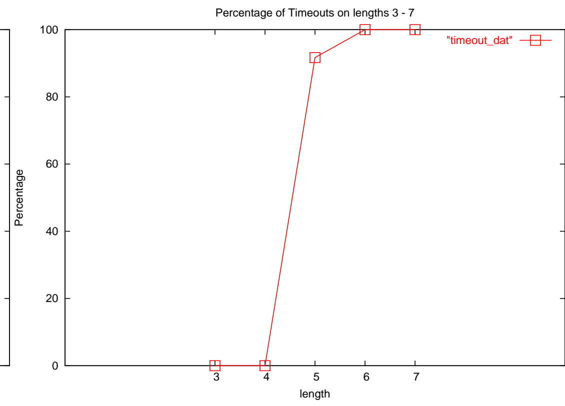


Fig. 4.18: Category G, Timeouts.

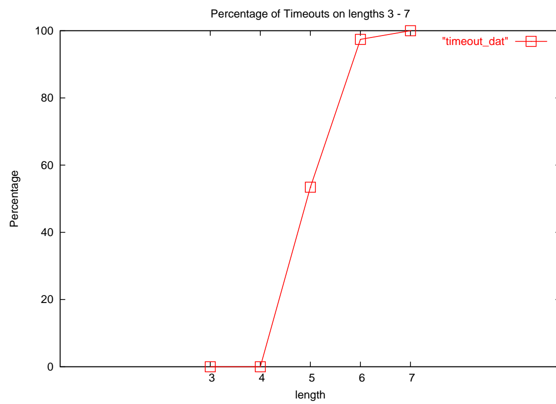


Fig. 4.19: Category H, Timeouts.

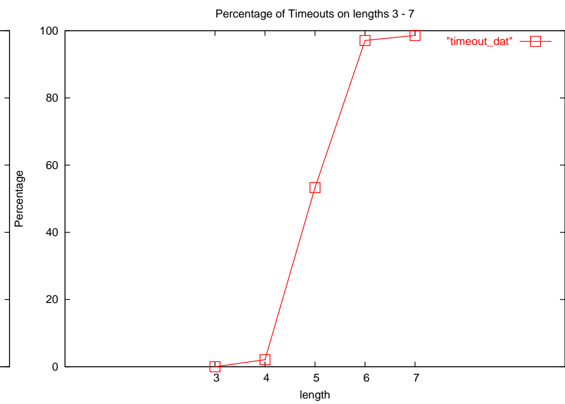


Fig. 4.20: Category I, Timeouts.