SAT Based Link-Grammar Parser

Filip Marić

August 13, 2008

Contents

1 Introduction

This document describes the theorethical and implementational aspects of a *Link Grammar Parser* based on SAT solving mehtods. The parser is developed as a *Google Summer of Code 2008. project*.

The Link Grammar Parser, initially developed at CMU, is a syntactic parser of English, based on link grammar — an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of a set of labeled links connecting pairs of words. The parser can also produce a "constituent" representation of a sentence (showing noun phrases, verb phrases, etc.). The link parser is in use in many software systems, including the AbiWord open- source word processor and RelEx Semantic Relation Extractor, and it has also been adapted for use in other languages. Therefore, having its faster implementation would be quite beneficial. The aim of this GSOC project was to describe a novel implementation of link parser based on the SAT/SMT solving. It has been hypothesized that a SAT/SMT version of the link parser may be an order of magnitude or more faster than the current version of the link parser, and if this hypothesis showed to be true, this could possibly make the link parser by far the fastest open-source full-sentence parser available.

1.1 Overview of the project and the main results

The project started on May 26. 2008. and suggested pencils-down date was August 11. 2008. During this period, several major milestones are accomplished and important conclusions have been made:

Feasibility. It is shown that it is possible to have a link-parser implementation completely based on SAT solving.

SAT encoding. Several SAT encodings of the link-grammar conditions are devised and described in this document. The descriptions are given in a strict framework of propositional logic. Many undocumented techniques used in the original link-parser implementation have also been described.

Implementation. All proposed encodings have been fully implemented and a fully functional parser based on SAT solving is implemented and available. The main web-page of the project is:

`http://www.matf.bg.ac.yu/~filip/gsoc`

It contains source code and documentation of several versions of the SAT based parser implementation.

Importance of conjunctive sentences. Experiments with the classic link-parser implementation show that the existing parser is very fast in parsing short and medium length sentences, but it performance gets worse when parsing very long sentences. Analysis shows that one specific kind of sentences poses much problems for the link parser — sentences that contain coordinating conjunctions. An example of such sentence is *"The cat and dog run."* and it is composed of two simpler sentences *"The cat runs."* and *"The dog runs."*. Since the classic implementation performes pretty poor on the conjunctive sentences, improving their parsing is an absolute priority for the overall parser efficiency.

Guiding. Developed theoretical models and implementation fail to outperform existing parser when it comes to enumerating all possible syntactically valid linkages. However, the experiments show that the novel SAT based implementation can significantly outperform classic implementation when it comes to finding just several valid linkages. Therefore, the SAT search has to be guided in a way that would ensure that the candidates for the semantically best linkages are among those several linkages that are found first. Several prototype guiding schemes are implemented, but serious guiding implementation is left for further work.

Dictionary. While working on the link-parser implementation it has been noted that the dictionary requires improvement. One step in this direction would to fix it so that syntactically correct sentences that are currently unparsable become parsable. The other step would be to extend the dictionary itself with the statistical information that would help in parser guiding.

In the rest of the text we will describe devised SAT encodings and their implementation.

1.2 Background

First we give some background information about link-grammars and SAT solving.

1.2.1 Link-grammars

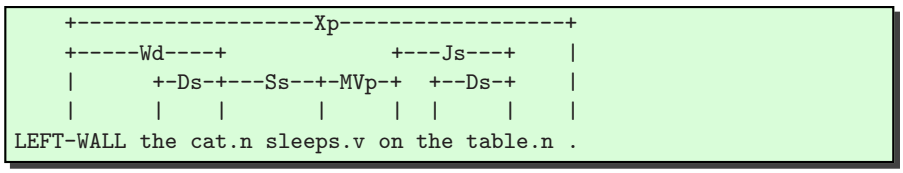
A *link-grammar* consists of a set of words (terminal symbols of the grammar) each of which has a list of attached connectors, some connecting to the left, and some connecting to the right. Two words of a sequence can be linked if there is a connector connecting the first word to the right and the second word to the left. A sequence of words is a valid sentence of the language defined by the grammar if there exists a way to draw links among its words satisfying the following conditions:

Satisfaction: the links satisfy the specific linking requirements of each word in the sequence.

Planarity: the links do not cross when drawn above the words.

Connectivity: the links suffice to connect all the words of the sequence together.

An example for a valid linkage of a sentence is:



Each connector has its specific usage and meaning. For example, the Ds connector connects determiners with singular nouns, MVp connects verbs with prepositions that start a verb-modifying phrase, etc.

1.2.2 SAT problem

Propositional satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. For example, the formula:

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

is true when x_1 is true and x_2, and x_3 are false.

It is a canonical NP-complete problem and it holds a central position in the field of computational complexity. SAT problem is also important in many practical applications such as electronic design automation, software and hardware verification, artificial intelligence, and operations research. Thanks to recent advances in propositional solving technology, SAT solvers are becoming the tool for attacking more and more practical problems.

Part I

Encoding

2
Word-tags and their satisfaction

The main characteristic of each word in the dictionary is its word-tag which describes all possible syntactically correct ways for this word to connect with other words.

2.1 Syntax of word-tag expressions

Word-tag expressions are built out of connectors (which are called atomic expressions), using the `&` and `or` operators. The operator `&` will denote the operation that will be called the *ordered strong conjunction* and the operator `or` will denote the operation that will be called *exclusive disjunction*. Although these connectors are in dictionary used as n-ary, it has been noted that after the dictionary parsing and expression simplification the resulting expression trees are always binary. This fact could be used for some minor optimizations, but it has been decide not to rely on this fact because it has not been documented.

2.2 Semantics of word-tag expressions

There were some differences between intended semantics of word-tag expressions and the description of semantics given in [1]. Finally, it has been agreed that the semantics is as it will be described here.

Consider the word-tag expression tree e corresponding to the word on position w_i.

Atomic expressions. If e is a connector, then it is satisfied if and only if there is a link attached exactly to this connector.

Note: all connectors in the tree are considered to be independent from one another, even if they have the same name and direction. Each connector can uniquely be identified by its word w_i and its unique position in the word-tag p_i (e.g., position in the pre-order tree traversal). Therefore, we will usually denote connectors by (w_i, p_i) pairs and implicitly assume their name and direction (e.g., C^+). Each link has two specific connectors that it connects (determined by their words and positions).

Ordered strong conjunction (`&`). If e is an "ordered strong conjunction" (`&`) of several subexpressions $e_1, \ldots, e_k$, it is satisfied only if all subexpressions $e_1, \ldots, e_k$ are satisfied, and if additional "order" constraints are satisfied. Order constraints demand that, if C' is a connector from the expression e_{i-1} and C'' is a connector from the expression e_i, then the words on which C' is attached must be strictly closer to the word w_i than words on which C'' is attached. Still, there is one very important additional condition that differentiates the `&` operator from ordinary conjunction. Namely, when the ordinary conjunction is not satisfied, one of its operands is not satisfied, but others can be. The ordered strong conjunction operation `&` imposes a much stronger condition. When a conjunction is false all of its operands must also be false which ensures that no connector in the expression e can be attached. This makes the `&` and $\wedge$ very, very different and it is one of the reasons why the conversion to SAT is not so direct and so promising as described in [1].

Exclusive disjunction (`or`). If e is and "exclusive disjunction" (`or`) of several subexpressions $e_1, \ldots, e_k$, it is satisfied only if exactly one of the expression $e_1, \ldots, e_k$ is satisfied. This directly corresponds to the xor operation $\oplus$.

2.3 Word-tag satisfaction encoding

One of the main linkage-correctness conditions is that all word-tag expressions should be satisfied. In this section the encoding of word-tag expression satisfaction conditions will be defined. This encoding resembles the standard Tseitin transformation used to convert an arbitrary propositional formula to CNF.

Let e be the word tag expression (or its subexpression) corresponding to the word w_i on position p_i in the sentence. A fresh variable will be assigned to the expression e. If it is the whole expression of the word-tag (expression on root position) the variable will be called simply w_i. If the expression is for example the second disjunct in the third conjunct of the whole word-tag expression the variable would be called $w_i\_c_3\_d_2$.

Link-grammars require that word-tags of every word in the sentence must be satisfied. The exception are so-called connective words and commas that can take a special coordination role that is going to be described in §5. For every word in a sentence (except these special words) a single literal clause which enforces that the whole word tag expression is satisfied is generated.

$$w_i$$

The main recursive procedure which generates satisfaction conditions for an expression will now be explained.

Atomic expression. If e is an atomic expression (i.e., a connector) its satisfaction conditions are generated using a specialized procedure whose behavior depends on whether the sentence contains conjunctions and therefore it will be explained in separate sections.

Ordered strong conjunction (`&`). Consider the case when e is an "ordered strong conjunction" (`&`) of several subexpressions $e_1, \ldots, e_k$. We distinguish the following three cases:

- $k = 0$, i.e., the empty conjunction. No conditions (i.e., clauses) are generated in this case, as its variable can be satisfied or not independent on the rest of the formula.
- $k = 1$, i.e., a single operand conjunction. Although this kind of expressions do not occur in the dictionary, they can be created during the word-tag expression simplification procedure. The formula can be simplified if these nodes are just "short-circuited". This is achieved by simply recursively applying the same procedure on the only subexpression e_1 with same parameters as for e itself.
- $k > 1$. This is the only case that generates some clauses. To each expression e_i, a fresh variable is assigned. The name v_c_i of this variable is built from the name v of the variable corresponding to the expression e, by appending the string $\_c_i$ (where i is a concrete number). It is now necessary to generate clauses that correspond to the condition:

  $$v \Leftrightarrow v\_c_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ v\_c_k$$

  This formula is converted to clauses using a specialized procedure[1]:

  $$\begin{aligned} v &\Rightarrow v\_c_1 \\ &\ldots \\ v &\Rightarrow v\_c_k \\ v\_c_1 &\Rightarrow v \\ &\ldots \\ v\_c_k &\Rightarrow v \end{aligned}$$

  Note: all variables v, v_c_1, ..., and v_c_k are mutually equivalent and can be replaced by a single variable while the listed clauses could be omitted. Still, that would make the implementation a bit harder but possibly would not bring so much benefit and that is why it is left for further work.

  When the clauses are generated, the ordering constraints are generated for each pair of consecutive subexpressions e_i and e_{i+1}. This procedure will also be described in a separate section.

Exclusive disjunction (`or`). Consider the case when e is an "exclusive disjunction" (`or`) of several subexpressions $e_1, \ldots, e_k$. We distinguish the following three cases:

- $k = 0$, i.e., the empty disjunction - this case should never occur.
- $k = 1$, i.e., a single operand disjunction. This case is completely equivalent to the case of conjunction with a single operand. As it

[1] These implications are trivially converted to clauses using the tautology $\neg p \vee q \equiv p \Rightarrow q$. Implications are used in presentation to improve readability.

was the case there, the tree is "short-circuited" by simply applying the recursive procedure to the only subexpression e_1 with the same parameters as for e itself.

- $k > 1$. This is the only case that generates some clauses. Again, to each expression e_i, a fresh variable is assigned. The name v_d_i of this variable is built from the name v of the variable corresponding to the expression e, by appending the string $\_d_i$ (where i is a concrete number). It is now necessary to generate clauses that correspond to the condition:

$$v \Leftrightarrow v\_d_1 \texttt{ or } \ldots \texttt{or } v\_d_k$$

This formula is converted to clauses using a specialized procedure:

$$\begin{gathered} v \;\Rightarrow\; v\_d_1 \;\vee\; \ldots \;\vee\; v\_d_k \\ v\_d_1 \;\Rightarrow\; v \\ \ldots \\ v\_d_k \;\Rightarrow\; v \end{gathered}$$

Since this disjunction must be exclusive, the exclusivity constraints are generated using the quadratic number of clauses. Since the expression tree is usually binary, this is not a problem, and only one clause is generated using this method.

$$\neg v\_d_i \;\vee\; \neg v\_d_j, \quad 1 \leq i < j \leq k$$

Implementation:

Implementation of this encoding is done in the class `SATEncoder` by the recursive method:

```
void SATEncoder::generate_satisfaction_for_expression(
      Exp* e, int wi, int& pi,
      char* var_name, int ancestor_cost);
```

Along with the expression `e`, word position `wi`, and current position connector `pi`, the name `var_name` of the variable that corresponds to the root of the expression `e` is passed along. This method also performs a cost (penalty) cut-off (see §2.5) and the parameter `ancestor_cost` is used for those purposes.

Connection between variable names and their numbers, converting expressions to CNF and generating clauses will be described in the part devoted to implementation (see §7, §8).

2.4 Conjunction order constraints

As it has been already said, one of the requirements for "ordered strong condition" is that for the expression $e_1 \,\&\, \ldots \,\&\, e_k$ on the word w_k, for each pair

of subexpressions (e_i, e_j), such that $i < j$ and e_i connects to the word w_i, e_j connects to the word w_j such that $\mathcal{B}(w_i, w_j, w_k)$ [2] i.e., it holds:

- If C_i^+ is a right pointing connector on e_i and C_j^+ is a right pointing connector on e_j, then there can be no words w_i and w_j such that C_i^+ is connected to w_i and C_j^+ is connected to w_j and $w_k < w_j < w_i$.
- If C_i^- is a left pointing connector on e_i and C_j^- is a left pointing connector on e_j, then there can be no words w_i and w_j such that C_i^- is connected to w_i and C_j^- is connected to w_j and $w_i < w_j < w_k$.

First, it can be noted that it suffices to impose this constraint only on adjacent subexpressions e_i and e_{i+1} and the rest would follow by transitivity. However, more optimizations can be made as the following example shows.

In the expression $(A_1^+ \texttt{ \& } (B_2^+ \texttt{ \& } B_3^-)) \texttt{ \& } (C_4^+ \texttt{ \& } (A_5^+ \texttt{ or } E_6^+ \texttt{ or } D_7^-))$, it suffices to require that the links on B_2^+ precede the links on C_4^+, that links on the A_1^+ precede the links on the B_2^+, and that links on C_4^+ precede the links on A_5^+ and precede the links on E_6^+. Also, it is required that links on B_3^- succeed the links on D_7^-. Then, all other ordering constraints would follow by transitivity.

Therefore, when comparing two adjacent subexpressions e_i and e_{i+1}, there is no need to check all their pairs of connectors pointing in some direction, but it suffices to compare only some of them. Several notions will now be introduced which will help identify pairs of connectors that can be skipped.

Denote by $empty(e, \pm)$ the fact that the expression e can be satisfied without using any $\pm$ sign connectors.[3] This can be recursively evaluated by the following procedure.

Atomic expressions. If the f is $C^{\pm}$ then $empty(f, \pm) = \bot$, and if it is $C^{\mp}$, then $empty(f, \pm) = \top$.

Ordered strong conjunctions (`&`). If f is $f_1 \texttt{ \& } \ldots \texttt{ \& } f_k$, then

$$empty(f, \pm) \Leftrightarrow empty(f_1, \pm) \wedge \ldots \wedge empty(f_k, \pm).$$

Exclusive disjunctions (`or`). If f is $f_1 \texttt{ or } \ldots \texttt{ or } f_k$, then

$$empty(f, \pm) \Leftrightarrow empty(f_1, \pm) \vee \ldots \vee empty(f_k, \pm).$$

Denote by $leading(f, \pm)$ all $^{\pm}$ connectors that can be the first connectors in a disjunct if the expression e was converted to DNF. This set can be recursively evaluated by the following procedure.

Atomic expressions. If the f is $C^{\pm}$ then $leading(f, \pm) = \{C^{\pm}\}$, and if it is $C^{\mp}$, then $leading(f, \pm) = \{\}$.

Ordered strong conjunctions (`&`). Let f be the $f_1 \texttt{ \& } \ldots \texttt{ \& } f_k$. Let $1 \leq l \leq k$ be the largest index such that

$$empty(f_1, \pm) \wedge \ldots \wedge empty(f_{l-1}, \pm) \wedge (l = k \vee \neg empty(f_l, \pm)).$$

Then

$$leading(f, \pm) = leading(f_1, \pm) \cup \ldots \cup leading(f_l, \pm)$$

[2] $\mathcal{B}(A, B, C)$ denotes that B is between A and C (i.e., either $A - B - C$ or $C - B - A$).
[3] This is a shorthand notation for $empty(f, +)$ and $empty(f, -)$.

Exclusive disjunctions (or). If f is f_1 `or` $\ldots$ `or` f_k, then

$$leading(f, \pm) = leading(f_1, \pm) \cup \ldots \cup leading(f_k, \pm)$$

Similarly, denote by $trailing(f, \pm)$ the set of all connectors that can occur as the last connector in a disjunct when f is converted to DNF. This set can be recursively evaluated by the following procedure.

Atomic expressions. If the f is $C^{\pm}$ then $trailing(f, \pm) = \{C^{\pm}\}$, and if it is $C^{\mp}$, then $trailing(f, \pm) = \{\}$.

Ordered strong conjunctions (&). Let f be the f_1 `&` $\ldots$ `&` f_k. Let $1 \leq l \leq k$ be the smallest index such that

$$empty(f_{l+1}, \pm) \wedge \ldots \wedge empty(f_k, \pm) \wedge (l = 1 \vee \neg empty(f_l, \pm)).$$

Then

$$trailing(f, \pm) = trailing(f_l, \pm) \cup \ldots \cup trailing(f_k, \pm)$$

Exclusive disjunctions (or). If f is f_1 `or` $\ldots$ `or` f_k, then

$$trailing(f, \pm) = trailing(f_1, \pm) \cup \ldots \cup trailing(f_k, \pm)$$

Having introduced these new notions, we can now describe how can the ordering constraints for the ordered conjunction be efficiently generated. Let f be the f_1 `&` $\ldots$ `&` f_k. It suffices to require that all links on connectors in $trailing(f_i, +)$ precede all links on connectors in $leading(f_{i+1}, +)$, for $1 \leq i < k$, and that all links on connectors in $trailing(f_i, -)$ succeed all links on connectors in $leading(f_{i+1}, -)$, for $1 \leq i < k$.

Let us look the given example once more. The ordering constraints are generated for each ordered strong conjunction node in the expression.

First such node is the whole expression $(A_1^+$ `&` $(B_2^+$ `&` $B_3^-))$ `&` $(C_4^+$ `&` $(A_5^+$ `or` E_6^+ `or` $D_7^-))$.

$trailing(A_1^+$ `&` $(B_2^+$ `&` $B_3^-), +)$ is $\{B_2^+\}$ and $leading(C_4^+$ `&` $(A_5^+$ `or` E_6^+ `or` $D_7^-), +)$ is $\{C_4^+\}$. Therefore links on B_2^+ must precede the links on C_4^+.

$trailing(A_1^+$ `&` $(B_2^+$ `&` $B_3^-), -)$ is $\{B_3^-\}$ and $leading(C_4^+$ `&` $(A_5^+$ `or` E_6^+ `or` $D_7^-), +)$ is $\{D_7^-\}$. Therefore, links on B_3^- must succeed the links on D_7^-.

Now consider the expression A_1^+ `&` $(B_2^+$ `&` $B_3^-)$

$trailing(A_1^+, +)$ is $\{A_1^+\}$ and $leading(B_2^+$ `&` $B_3^-, +)$ is $\{B_2^+\}$. Therefore links on A_1^+ must precede the links on B_2^+. Since $trailing(A_1^+, -)$, $leading(B_3^-, +)$, and $trailing(B_2^+, -)$ are empty there are no more constraints for this expression.

Now consider the expression C_4^+ `&` $(A_5^+$ `or` E_6^+ `or` $D_7^-)$.

$trailing(C_4^+, +)$ is $\{C_4^+\}$, and $leading(A_5^+$ `or` E_6^+ `or` $D_7^-, +)$ is $\{A_5^+, E_6^+\}$. Therefore, links on C_4^+ must precede the links on A_5^+ and on E_6^+. Since, $trailing(C_4^+, -)$ is empty, no more constraints are needed for this formula.

In the rest of this section it will be assumed that for each connector $(w_i, p_i$ and each word w_j, a variable $link_{cw}((w_i, p_i), w_j)$ has been defined and that it denotes that connector (w_i, p_i) has made a link with the word w_j.

When pairs of connectors (e.g., (w_k, p_i') and (w_k, p_i''), $pi' < pi''$) for which the ordering conditions should be generated are identified, the following clauses are generated.

$$\neg link_{cw}((w_k, p_i'), w_i) \vee \neg link_{cw}((w_k, p_i''), w_j), \quad \mathcal{B}(w_i, w_j, w_k)$$

Implementation:

Order constraints are generated using the method

```
void SATEncoder::generate_conjunction_order_conditions(
        int wi, int pi, Exp* e1, Exp* e2);
```

Leading and trailing connectors are detected using methods

```
void trailing_connectors(
        char dir, Exp* exp,
        int w, int& dfs_position,
        std::vector<PositionConnector*>& connectors);

void leading_connectors(
        char dir, Exp* exp,
        int w, int& dfs_position,
        std::vector<PositionConnector*>& connectors);
```

All matches of a connector are easily found using the `WordTag` datastructure (see 9.3).

2.5 Cost cut-off

Word tag expressions in the dictionary are sometimes explicitly assigned small natural numbers (usually 1, 2 or 3) that are called *costs* or *penalties*. All expressions that do not have explicitly assigned costs are considered to have cost 0. Costs are used to denote how likely it is that some parts of word-tags are used in valid linkages. Higher the cost for a node is, it is more unlikely that that node is used in a valid linkage. Costs are inherited along the word-tag expression tree. The syntax for specifying costs suggests that the number of brackets surrounding a connector determine its cost. So, for example, the expression `A+ or [B- & [[C+]] & D-]` denotes that the connector A^+ has the cost 0, connectors B^- and D^- have the cost 1, and the connector C^+ has the cost 3. A simple cut-off rule says that connectors with the cost higher then a threshold value (usually 2) should not be used while parsing. When it is determined that the cost of a node in the word-tag expression tree (whose name is e.g., v) is higher then the threshold value, the single literal clause

$$\neg v$$

is generated.

Implementation:

Cost cut-off is performed during the recursive expression tree traversal in the `generate_satisfaction_for_expression` method. The parameter `ancestor_cost` passes accumulated cost of all ancestor nodes of the current node. It cost is summed with the cost of its ancestor and if that number exceeds the threshold, the current node is cut-off. In that case, there is no need to process its descendants and generate satisfaction conditions for them (this is one of the main reasons why the cost cut-off is done in parallel with satisfaction condition generating).

3 Global constraints

3.1 Planarity

One of the constraints of the link-grammars is that no links can cross. Once the $linked(w_i, w_j)$ are defined this conditions can be defined by a $O(n^4)$ number of clauses.

$$\neg linked(w_{i_1}, w_{i_2}) \vee \neg linked(w_{j_1}, w_{j_2}), \quad w_{j_1} < w_{i_1} < w_{j_2} < w_{i_2},$$
$$\neg linked(w_{i_1}, w_{i_2}) \vee \neg linked(w_{j_1}, w_{j_2}), \quad w_{i_1} < w_{j_1} < w_{i_2} < w_{j_2},$$

Usually, links are not possible between all the words in a sentence, and it is known that a huge number of $linked(w_i, w_j)$ variables cannot be satisfied. The clauses that contain those variables are trivially satisfied and therefore can be omitted. Variables $linked(w_i, w_j)$ for which a link is not possible are not present in the resulting CNF formula.

One possible way to reduce the number of link-crossing clauses is to introduce variables $not\_linked(w_i, w_j)^+$ and $not\_linked(w_j, w_i)^-$ which would denote that the word w_i cannot be linked to words that are on the right or on the left of the word w_j respectively. This technique is not implemented.

Implementation:

The links-crossing constraints are generated using the method:

```
void SATEncoder::generate_planarity_constraints();
```

In order to reduce the number of clauses, the information about what pairs of words w_i and w_j can be possible is stored in a matrix `_linked_possible`. This matrix is consulted when clauses are generated. This matrix is populated when $linked(w_i, w_j)$ variables are defined in the method `generate_linked_definitions`. This method is going to be defined separately in §4 and §5.

3.2 Connectivity

One of the linkage-correctness requirements is that linkages have to be connected, i.e., each word should be reachable starting from the left wall (imaginary word that precedes the first word of the sentence). Although a lot of attention has been put on different algorithms which would impose the connectivity requirements during the SAT search, all SAT encodings of this condition showed out to be very complex. The approach described in [1] was both incorrect and inefficient. A correct, but still inefficient encoding was proposed and even implemented, but it turned out that the best way to check for connectivity is aposteriori. Using this approach, connectivity is checked only after a linkage has been constructed.

The only type of connectivity constraints that is asserted before the search starts is so called *weak-connectivity* which requires that all words in a sentence are linked to at least one other word. These constraints are imposed through the clauses[1]:

$$\bigvee_{w_j<w_i} linked(w_j, w_i) \vee \bigvee_{w_i<w_j} linked(w_i, w_j)$$

At it was the case with link-crossing constraints (see §3.1), some of the $linked(w_i, w_j)$ variables are not defined at all because corresponding words cannot be linked in any way. These variables are eliminated from the weak-connectivity clauses during their construction.

The connectivity graph is reconstructed from the values of $linked(w_i, w_j)$ variables which denote whether the words w_i and w_j are linked by some kind of link. Once the graph is constructed, a simple DFS based procedure is used to enumerate all connected components of the graph. If there is only a single connectivity component, the graph is connected and the linkage passes the connectivity test. If there are several connectivity components, special clauses are generated and added to the model in order to prevent the same kind of dis-connectivity for the future linkages. These clauses are constructed based on the requirement that each of the connectivity components should have at least one branch that connects it with nodes that are not in that component. In that way, for each connectivity component C_k the clause

$$\bigvee_{w_i \in C_k, w_j \notin C_k} linked(w_i, wj)$$

is generated and added to formula. If there are exactly two connected components (which is very often the case), it suffices to add only the clause for one of them (the other clause would be exactly the same). It turns out that these clauses have very good search-space pruning power as they cut-off not only this single disconnected linkage, but also they cut-off all linkages that would be disconnected in a way which would have a same connected component as this linkage does.

[1] Recall that $linked(w_i, w_j)$ variables require that $w_i < w_j$. That is the reason why two groups of literals are separated.

These clauses are inconsistent with the current linkage and therefore are conflicting clauses. When the first of them is added to the SAT solver, it backtracks and resolves the conflict. When other clauses are added, solver has already backtracked and these clauses can be still conflicting clauses, they can be unit clauses or ordinary clauses. Therefore, the MiniSAT procedure of adding clauses had to be altered and implemented so that it handles all these cases properly.

Implementation:

Before the search starts, weak connectivity constraints are generated by using:

```
void SATEncoder::generate_weak_connectivity();
```

Once the linkage is constructed, its connectivity is checked by using:

```
bool SATEncoder::check_connectivity(
            std::vector<int>& components);
```

This method assumes that the MiniSAT (represented by the class `Solver`) has successfully constructed a linkage. It uses functionality of the class `Variables` (see §7) to find all $linked(w_i, w_j)$ variables and consults the `Solver` to check which of them are satisfied. When the connectivity graph is constructed in this way, a DFS based connectivity component analysis is performed by the method `dfs_connectivity_components`. Finally, if linkage is disconnected and several components are identified, the clauses that prohibits the same type of partitioning and generated by using:

```
void generate_partition_prohibiting(
     const std::vector<int>& components);
```

3.3 Post-processing

PP pruning. The most frequently used *"contains one rules"* require that if there is one kind of link label occurring in the sentence there must be some other type of link occurring in the same domain as the first one. For example the rule `Wq, SI SFI SXI` requires that if there is a Wq link label in the sentence, there should be an SI, SFI or SXI link label in its domain. The label Wq is called *trigger*, and the labels SI, SFI and SXI are called criterion links. When the "in its domain" part of requirement is removed, we get a weaker condition which could easily be encoded in SAT by propositional clauses. For each "contains one rule", the set T of $link_{cc}((w_i, p_i), (w_j, p_j))$ variables that match the trigger is identified, and the set C of $link_{cc}((w_i, p_i), (w_j, p_j))$ variables that match some of the criterions is identified. Then the following clauses are generated:

$$t \Rightarrow \bigvee_{c \in C} c, \qquad t \in T$$

Implementation:

Since it is assumed that word-tag satisfaction conditions are already generated, all $link_{cc}((w_i, p_i), (w_j, p_j))$ can be easily enumerated using the functionality of the class `Variables` (see §7). Check if a variable matches a trigger or a criterion label is done using the existing functionality from the classical link-parser implementation.

4
Conjunction free sentences.

In this chapter we will describe the SAT encoding of the link-parsing conditions for sentences that do not contain conjunctions. When the sentence does not contain a so called *connective word* (e.g., "and", "or", "but"), the only kind of connections that are possible are direct connections between two connectors.

Satisfaction of connectors. Let v be the variable that corresponds to node of the word-tag expression tree that contains this connector. The variable v encodes the fact that the connector (w_i, p_i) is satisfied. Let W be a collection of connectors that match (w_i, p_i) and let $link_{cc}((w_i, p_i), (w_j, p_j))$ be a collection of variables that encode that (w_i, p_i) is connected to the connector $(w_j, p_j) \in W$. Then the following condition must hold:

$$v \Leftrightarrow \bigvee_{(w_j,p_j)\in W} link_{cc}((w_i, p_i), (w_j, p_j)),$$

if (w_i, p_i) is a multi-connector or

$$v \Leftrightarrow \bigoplus_{(w_j,p_j)\in W} link_{cc}((w_i, p_i), (w_j, p_j)),$$

if it is not.

Implementation:

Satisfaction condition for connector are generated by using:

```
void SATEncoder::generate_satisfaction_for_connector(
      Exp* e, int wi, int& pi,
      char* var_name, int ancestor_cost);
```

All matches for the connector (w_i, p_i) are detected by a simple lookup information since the matches are all cached in the `WordTag` data structures (`_word_tag[wi].get_connector(pi).matches`) (see §9.3). Based on this information, a collection of variables $link_{cc}((w_i, p_i), (w_j, p_j))$ is simply formed.

Checking if the two words are linked. Two words w_i and w_j, such that $w_i < w_j$ are linked and $linked(w_i, w_j)$ holds if and only if there are two connectors (w_i, p_i) and (w_j, p_j) such that $link_{cc}((w_i, p_i), (w_j, p_j))$ holds. Therefore, the following condition must hold.

$$linked(w_i, w_j) \Leftrightarrow \bigoplus_{p_i, p_j} link_{cc}((w_i, p_i), (w_j, p_j))$$

Since there cannot be two different links between the two words, the disjunction must be exclusive.

Implementation:

$linked(w_i, w_j)$ variables are defined by using:

```
void SATEncoder::generate_linked_definitions();
```

The detection of candidate variables $link_{cc}((w_i, p_i), (w_j, p_j))$ is very fast, once the word tags are precompiled and stored in `WordTag` datastructure (see 9.3). Iteration is done through the vector `_word_tags[wi].get_right_connectors()` and all matches with the word w_j are collected.

5 Conjunctive sentences

5.1 Introduction to fat-links

Although the classical link-parser implementation uses fat-links, there is not much text about them in the documentation. In this section fat-link approach to handling conjunctive sentences will be described.

Every connector in a sentence that contains a *conjunctive word* ("and", "or", etc.) can be satisfied in the usual way, by a direct link with another connector, or can be satisfied in an indirect way, through a connective word (or comma) to which it is linked by a special kind of link called the *fat-link*. Fat-links are established between words and do not use any connectors. Fat-links are also considered to be directed links, and it is said that one of the words linked is down and the other is up. When connective words (or commas) are used in this special kind of way, they have to have two fat-links *pointing down* to two words; one on its left hand side and the other from its right hand side. We say that the fat-links on these two words *point up* to the connective word (or comma). These words can again be either ordinary words or special connective words (or commas), and so the fat-links of a sentence form a binary tree forest structure. Words that are linked up to the connective word (or comma) by a fat-link can also use ordinary thin-links, but these thin-links cannot be attached to words beyond the fat-link, i.e. the connective word (or comma) has to be the word that is furthest linked to the left or to the right.

When no fat-links are present in a sentence, the only way to satisfy a connector (e.g., (w_i, p_i)) is by establishing a direct link with another connector. When fat-link trees are present in a sentence, for some connector types[1] connector can be connected to not only directly to another connector, but also to a connective word w_j which is on the top (or somewhere in the middle) of the fat-link tree and therefore has two fat-links pointing down. If an ordinary word w_k is fat-linked up to this connective word w_j, there must be a connector (w_k, p_k) that matches the connector (w_i, p_i). In this case, it is considered that an indirect link is established between these two connectors. When both words (e.g., w_{k_1} and w_{k_2}) fat-linked up to the special connective word w_j are ordinary

[1] The connectors that can directly attach to connective words in the fat-link trees are called *andable connectors* and are listed in the dictionary file.

words, it can happen that they both have connectors (w_{k_1}, p_{k_1}) and (w_{k_2}, p_{k_2}) matching the connector (w_i, p_i), but the labels established are incompatible. For example, the connector C^+ matches both Cx^- and Cy^-, but labels are Cx and Cy which are incompatible. This case is considered to be invalid and it is ruled out by the link-grammar constraints. Since words in a fat-linked tree are organized in a hierarchical way, a word w_k that is fat-linked up to the special connective word w_j can be a special connective word itself. In that case it must have two words fat-linked up to it and to which the same rules apply.

Conjunctive words also have their dictionary entries and can also act as ordinary words. Still, one of the required restrictions is that connective words cannot act as ordinary and as special in the same time, i.e., connective words can use connectors in their word-tags iff they do not have fat-links down. In either case, they can have a fat-link pointing up to another connective word. Commas are also treated as special kind of connective words, but rules for them are a little bit different.

1. Commas can act as special conjunctive words and in that case they have to use exactly three fat-links, one pointing down to the left, one pointing down to the right, and one pointing up to the right in the fat-link tree.

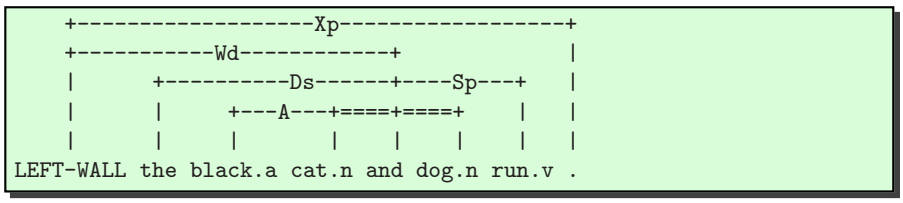
2. Ordinary conjunctive words ("and", "or", etc.) when acting as special words, have to have two fat-links, one pointing down to the left and one pointing down to the right, and they can use one additional fat-link that points up the fat-link tree (it can be either to the left, or to the right).

3. Ordinary words can only be leaves of the fat-link trees, and they can use exactly one fat link pointing up (either to the left, or to the right).

Let us illustrate all this on several examples. The following linkage contains a coordinating conjunction structure and first we present it by two separate sublinkages which is a standard way of displaying linkages containing conjunctions in the classic link-parser implementation.

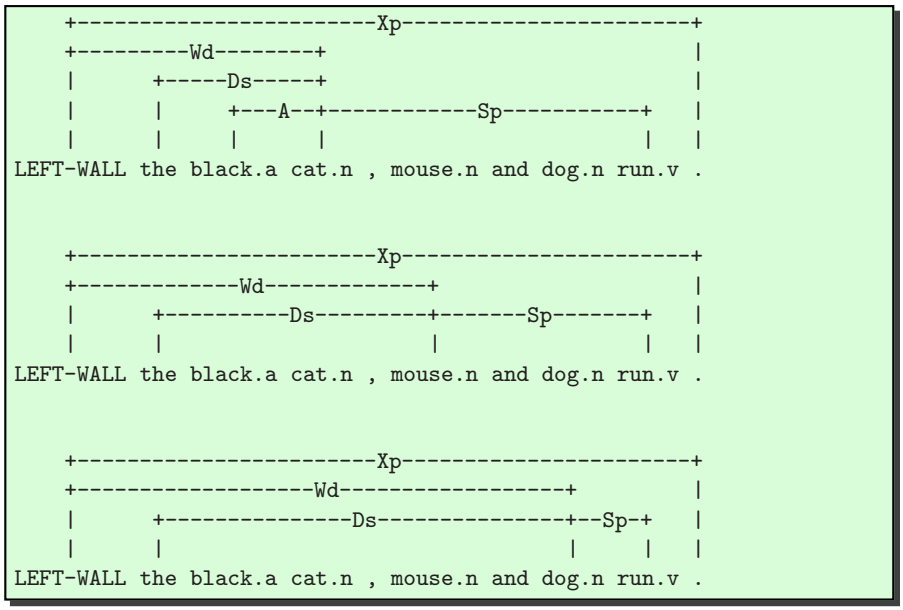
```
    +-------------------Xp------------------+
    +---------Wd--------+                   |
    |      +-----Ds-----+                   |
    |      |     +---A--+-------Sp------+   |
    |      |     |      |               |   |
LEFT-WALL the black.a cat.n and dog.n run.v .


    +-------------------Xp------------------+
    +--------------Wd-------------+         |
    |      +----------Ds----------+--Sp-+   |
    |      |                      |     |   |
LEFT-WALL the black.a cat.n and dog.n run.v .
```

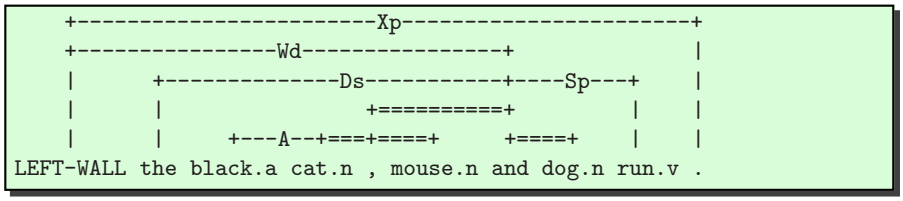
This linkage is internally represented as:



In this case, the words cat and dog are connected with the conjunctive word "and" by using fat-links. Connector D^+ other word "the" is attached to the connective word "and". Each of the two words that are its children in the fat-link tree ("cat" and "dog") has the connector "Ds-" which is then considered to be attached to the connector "D+" on the word "the". The similar case happens with the connector Sp^- on the word "run". Notice that although the word "cat" makes an indirect connector to the word "the" it can still have direct links, as it is the case with the link A between words "black" and "cat".

Here is a bit more complicated example that uses fat-link nesting.





The fat link tree structure is as follows:

5.2 Fat-link conditions - encoding

Fat-link variables. The fact that the word w_i is connected up to a connective word w_j is going to be described by a variable $fat\_link(w_i, w_j)$. The walls cannot have fat links up, so these variables will be defined when it holds $0 < w_i < n-1$ and $conn\_comma(w_j)$. The predicate $conn\_comma$ will denote that the word w_j is either a connective word ("and", "or", etc.) or a comma.

Fat-link down existence variables. For each connective word w_i we will introduce $w_i^{fl\_d}$ variable which denotes that the word w_i has fatlinks down. The following conditions define these variables:

$$w_i^{fl\_d} \Leftrightarrow \bigoplus_{0<w_j<w_i} fat\_link(w_j, w_i)$$

$$w_i^{fl\_d} \Leftrightarrow \bigoplus_{w_i<w_j<n-1} fat\_link(w_j, w_i)$$

The n denotes the length of the sentence. Note that the walls cannot have fat-links up. These definitions ensure that if the connective word has fat-links down, it has exactly two fat-links down — one to the left and one to the right.

Connectives can serve either as special or ordinary words. The condition that a connective word cannot serve both as an ordinary word and as a connective word is encoded using:

$$w_i \oplus w_i^{fl\_d}$$

If we recall that the variable w_i denotes that the word-tag expression of the word w_i is satisfied (as it was defined in Section 2.3), this condition requires that either word-tag of the word w_i is satisfied, or it has fat-links down, but both things together cannot happen.

Fat links up definitions are similar. For each word w_i (except the walls, i.e., $0 < w_i < n-1$), we introduce the variable $w_i^{fl\_ul}$ which denotes that the word has fat-links up to the left, the variable $w_i^{fl\_ur}$ which denotes that the word has fat-links up to the right.

$$w_i^{fl\_ul} \Leftrightarrow \bigoplus_{0<w_j<w_i,\ conn\_comma(w_j)} fat\_link(i, j)$$

$$w_i^{fl\_ur} \Leftrightarrow \bigoplus_{i<w_j<n-1,\ conn\_comma(w_j)} fat\_link(i, j)$$

$$\neg w_i^{fl\_ul} \wedge \neg w_i^{fl\_ur}$$

The special conditions that are placed on the commas are encoded with:

$$\neg w_i^{fl\_d} \vee \neg w_i^{fl\_ul}$$
$$\neg w_i^{fl\_d} \vee w_i^{fl\_ur}$$

where it is assumed that w_i is a comma i.e., $comma(w_i)$ holds.

If a connective word (or comma) w_i has fat-link down to the word w_j (i.e., $fat\_link(w_j, w_i)$, it cannot have fat-links up between w_j and w_i. If $w_j < w_i$, this condition is encoded with:

$$\neg fat\_link(w_j, w_i) \vee \neg fat\_link(w_i, w_k), \quad w_j \leq w_k < w_i,\ conn\_comma(w_k),$$

and if $w_i < w_k$, then with:

$$\neg fat\_link(w_j, w_i) \vee \neg fat\_link(w_i, w_k), \quad w_i < w_k \leq w_j,\ conn\_comma(w_k),$$

The requirement that there can be no links beyond the fat-link up is encoded in the following way. If there is a fat-link from the word w_j up to the word w_i (i.e., $fat\_link(w_j, w_i)$), then if $w_j < w_i$

$$\neg fat\_link(w_j, w_i) \vee \neg linked(w_j, w_k), \quad w_i < w_k,$$

and if $w_i < w_j$, then

$$\neg fat\_link(w_j, w_i) \vee \neg linked(w_k, w_j), \quad w_k < w_i,$$

Variable $linked(w_i, w_j)$ where $w_i < w_j$ denotes that there is some kind of link (either thin or fat) between words w_i and w_j. There definition depends on the specific encoding used and going to be explained in the rest of the text.

5.3 Different link types - examples

As in conjunction-free case, direct links can be established between connectors. Unfortunately, in the presence of fat-links connectors, direct connections can be made from a connector to a conjunctive word, or, even worse, from one conjunctive word to another. Even when a connector (w_i, p_i) is directly attached to a special connective word w_j, words below the word w_j that are fat-linked up to it must have connectors that match the connector (w_i, p_i) and connector (w_i, p_i) is indirectly attached to those connectors.

The SAT encoding encoding must be extended in a way which supports all these different kinds of links. During the course of this projects, two different encodings were developed and implemented. We will describe them in the following two subsections.

5.3.1 Several connector-connector link types

Since there are three different types of direct connections that can be made in a conjunctive sentence, the following three types of variables represent the basic variables of the encoding.

1. $link_{cc}^{dd}((w_i, p_i), (w_j, p_j))$ - two connectors (w_i, p_i) are directly connected.
2. $link_{cw}^{dd}((w_i, p_i), w_j)$ - connector (w_i, p_i) is directly connected to connective word w_j.
3. $link_{ww}^{dd}(w_{j_1}, w_{j_2})$ - connective words w_{j_1} and w_{j_2} are directly connected.

Along with this basic variables that describe direct connections, the following implied variables that describe indirect connections are introduced. This variables are then used to encode various linkage correctnes constraints.

1. $link_{cw}^{di}((w_i, p_i), w_j)$ - The connector (w_i, p_i) is indirectly connected to a word w_j i.e., it is directly connected to a connective word w_k and the word w_j is below w_k in the fat-link tree.
2. $link_{cc}^{di}((w_i, p_i), (w_j, p_j))$ - a connector (w_i, p_i) is indirectly connected to the connector (w_j, p_j) i.e., it is directly connected to a connective word w_k and the word w_j (on which the connector (w_j, p_j) lies) is below w_k in the fat-link tree.
3. $link_{ww}^{di}(w_{j_1}, w_{j_2})$ - The connective word w_{j_1} is indirectly connected to word w_{j_2} i.e., it is directly connected to a connective word w_k and the word w_{j_2} is below w_k in the fat-link tree.
4. $link_{cw}^{ii}((w_i, p_i), w_j)$ - the connector (w_i, p_i) is indirectly-indirectly connected to the word w_j, i.e., there is a direct connection betwen two connective words w_{k_1} and w_{k_2} and word w_i is below w_{k_1} and w_j is below w_{k_2} in the fat-link tree.
5. $link_{cc}^{ii}((w_i, p_i), (w_j, p_j))$ - the connector (w_i, p_i) is indirectly-indirectly connected to the connector (w_j, p_j) i.e., there is a direct connection betwen two connective words w_{k_1} and w_{k_2} and word w_i is below w_{k_1} and w_j is below w_{k_2} in the fat-link tree.
6. $link_{ww}^{ii}(w_{j_1}, w_{j_2})$ - The connective word w_{j_1} is indirectly-indirectly connected to a word w_{j_2} i.e., there is a direct connection betwen two connective words w_{k_1} and w_{k_2} and word w_{j_1} is below w_{k_1} and w_{j_2} is below w_{k_2} in the fat-link tree.

In order to model sentences that do not contain direct-links between two connective words ("and-and" sentences), it suffices to define only $link_{cc}^{dd}((w_i, p_i),$-$(w_j, p_j))$, $link_{cw}^{dd}((w_i, p_i), w_j)$, $link_{cw}^{di}((w_i, p_i), w_j)$, and $link_{cc}^{di}((w_i, p_i), (w_j, p_j))$ variables. This approach has been implemented and gave very promissing results. Unfortunately, it has been noted that it does not support "and-and" sentences. In order to have support for "and-and" sentences, all other listed variables have to be defined. This would yield very big formula with very high number of variables. That is why this encoding has not been fully implemented, but an alternative encoding that gives smaller formulae is developed. We describe this alternative encoding in the following section.

5.3.2 Single connector-connector link type

The SAT encoding that is going to be described in this section is based on the fact that each satisfied connector is linked to one or more connectors on the other words. If we know what links between connectors are established, then we can "calculate" what connections are direct, and what connections are indirect, what words are linked with thin links etc. Therefore, for basic variables of the system we only take variables that describe established links between connectors (regardless of whether these links are direct, indirect, or indirect-indirect speaking in terms of the previous encoding). We will try to illustrate this encoding through an example and along these basic variables, we will introduce several implied types of variables. However the number of these variables, as well as the number of clauses used for their definition and for formulation of valid-linkage conditions is much smaller than in the previous encoding. The implementation of this new encoding is implemented, but it has not yet been fine-tuned so it does not give as promissing results as the previous one does.

Consider the following simplified example (connections with the left-wall will not be discussed for this example).

```
      +----Sp----+
 +====+====+     |
 |    |    |     |
cats and  dogs  run
 1    2    3     4
```

In this case, the connector Sp^- on the word "run" connects both to the connector Sp^+ on the word "cats" and the connector "Sp+" on the word "dogs".

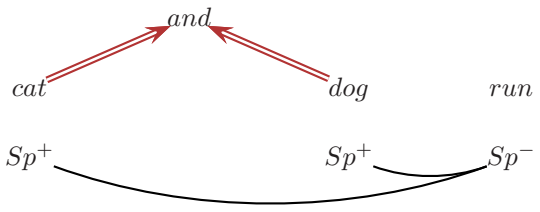
The fat-link structure of this sentence is described by

$$fat\_link(1,2)$$
$$fat\_link(3,2)$$

The fact that there is a link between the connector in word w_i on position p_i and the connector w_j on position p_j will be encoded by the variable $link_{cc}((w_i,p_i),(w_j,p_j))$. In the given example, we would say that following links between connectors are established (instead of positions, for better readability connector names are written):

$$link_{cc}((1,Sp^+),(4,Sp^-))$$
$$link_{cc}((3,Sp^+),(4,Sp^-))$$

This can be illustrated by:

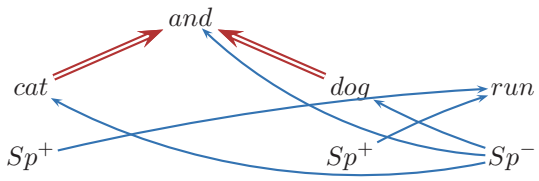


Apart from these connector-connector connections, we will say that, for example, connector Sp^- makes connections to words 1, 2, and 3. Connections between connectors and words will be denoted by variables $link\_cw((w_i,p_i),w_j)$.

In the given example, the following connector-word connections exist:

$$link_{cw}((1, Sp^+), 4)$$
$$link_{cw}((3, Sp^+), 4)$$
$$link_{cw}((4, Sp^-), 1)$$
$$link_{cw}((4, Sp^-), 2)$$
$$link_{cw}((4, Sp^-), 3)$$

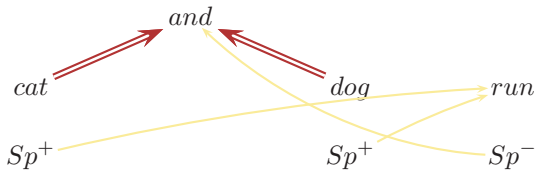
This can be illustrated with:



Connections of connector Sp^- to words 1 and 3 will be called *indirect connections*, while the connection with the word 2 will be called a *direct connection*. A connector-word connection will be called a direct connection iff it is not inherited from the word above it in a fat-link tree. Variables $link\_top_{cw}((w_i, p_i), w_j)$ will denote this kind of connection[2]. In the previous example, the following direct connections are made:

$$link\_top_{cw}((1, Sp^+), 4)$$
$$link\_top_{cw}((3, Sp^+), 4)$$
$$link\_top_{cw}((4, Sp^-), 2)$$

This can be illustrated with:



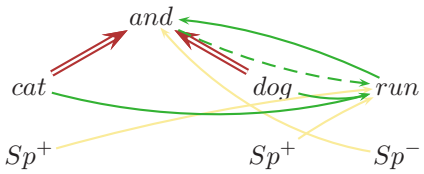
If connector is not a multi-connector, it should be able to make only one direct connection, but can make several indirect connections, provided that the words it connects to are all parts of a single fat-link tree.

In order to detect direct links between words, we have to introduce direct connections from words to words. Therefore, the variables $link\_top_{ww}(w_i, w_j)$ will denote that there is a direct connection from the word w_i to the word w_j. This connection must not be inherited from above on the w_j side, i.e., w_j is either top of fat-link tree or, if word w_j has a fat-link parent w_k, then there is no connection from the word w_i to the word w_k. In the given example, the following $link\_top_{ww}$ connections hold:

$$link\_top_{ww}(1, 4)$$
$$link\_top_{ww}(2, 4)$$
$$link\_top_{ww}(3, 4)$$
$$link\_top_{ww}(4, 2)$$

[2]The name *link_top* is used to indicate that the link is made to the top of a fat-link structure.

This can be illustrated with



If w_i is used as an ordinary word (i.e., it has no fat-links down), it is directly connected to the top of the word w_j iff it has a connector that is directly connected to the top of word w_j. If it is used as a connective word (i.e., it has fat-links down), then it directly connected to the top of the word w_j iff both of its children in the fat-link tree are directly connected to the word w_j. The connection from "and" to "run" drawn in dashed line-style is formed this way.

Finally, there is a thin-link between two words iff they are connected on top of one another. The fact that there is a thin-link between two-words is denoted by variables $thin\_link(w_i, w_j)$, where $w_i < w_j$. In the previous example, the only thin link is between "and" and "run":

$$thin\_link(2, 4)$$

We would say that two variables are linked, denoted by $linked(w_i, w_j)$, where $w_i < w_j$, iff there is either a thin-link between them, or if one (or both) of them is a connective word then a fat-link up to it.

This model is good enough to describe even more complex cases when there is direct link between two connective words. One of such sentences example in the sentence "the cat and dog run and jump". The only valid linkage for this sentence is:

```
+-----Ds---+------Sp------+
|     +====+====+    +====+====+
|     |    |    |    |    |    |
the  cat  and  dog  run  and  jump
 1    2    3    4    5    6    7
```

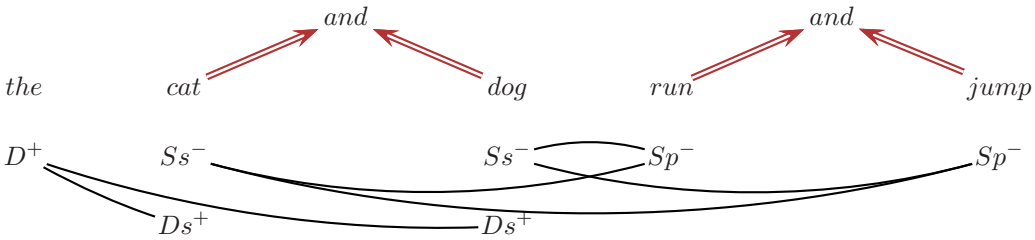
The fat-link structure of the sentence is described by:

$$fat\_link(2, 3)$$
$$fat\_link(4, 3)$$
$$fat\_link(5, 6)$$
$$fat\_link(7, 6)$$

The links that are established between connectors are:

$$link_{cc}((1, D^+), (2, Ds^-))$$
$$link_{cc}((1, D^+), (4, Ds^-))$$
$$link_{cc}((2, Ss^+), (5, Sp^-))$$
$$link_{cc}((4, Ss^+), (5, Sp^-))$$
$$link_{cc}((2, Ss^+), (7, Sp^-))$$
$$link_{cc}((4, Ss^+), (7, Sp^-))$$

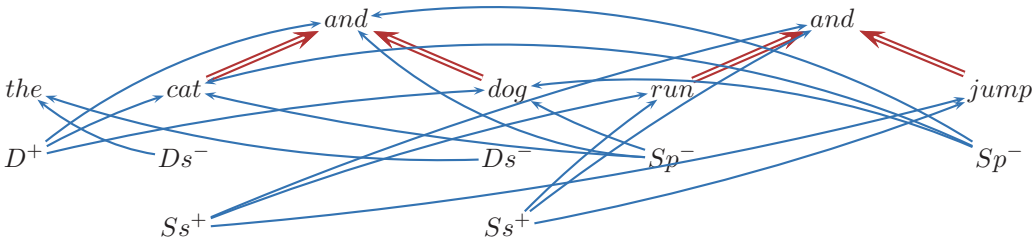
This can be illustrated by:



Using only the links established between connectors, links from connectors to words can be determined.

$$
\begin{gathered}
link_{cw}((1, D^+), 2) \\
link_{cw}((1, D^+), 3) \\
link_{cw}((1, D^+), 4) \\
link_{cw}((2, Ds^-), 1) \\
link_{cw}((4, Ds^-), 1) \\
link_{cw}((2, Ss^+), 5) \\
link_{cw}((2, Ss^+), 6) \\
link_{cw}((2, Ss^+), 7) \\
link_{cw}((4, Ss^+), 5) \\
link_{cw}((4, Ss^+), 6) \\
link_{cw}((4, Ss^+), 7) \\
link_{cw}((5, Sp^-), 2) \\
link_{cw}((5, Sp^-), 3) \\
link_{cw}((5, Sp^-), 4) \\
link_{cw}((7, Sp^-), 2) \\
link_{cw}((7, Sp^-), 3) \\
link_{cw}((7, Sp^-), 4)
\end{gathered}
$$

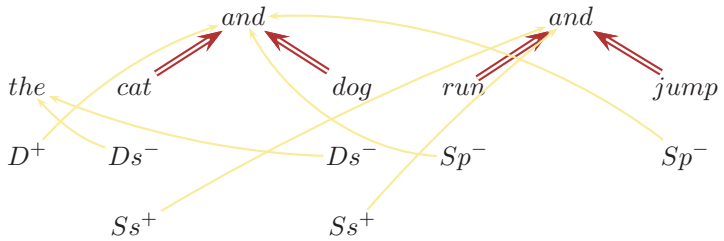
This can be illustrated by:



Only some of these links are “direct” meaning that they connect to top of a potential fat-link tree (to be more precise, they do not connect somewhere bellow the root of a fat-link tree).

$$
\begin{gathered}
link\_top_{cw}((1, D^+), 3) \\
link\_top_{cw}((2, Ds^-), 1) \\
link\_top_{cw}((4, Ds^-), 1) \\
link\_top_{cw}((2, Ss^+), 6) \\
link\_top_{cw}((4, Ss^+), 6) \\
link\_top_{cw}((5, Sp^-), 3) \\
link\_top_{cw}((7, Sp^-), 3)
\end{gathered}
$$

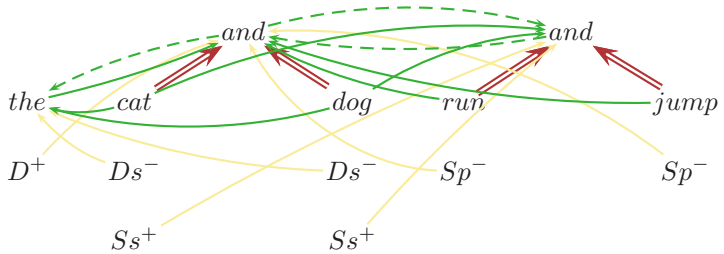
This can be illustrated by:



Using this, the "direct" connections from words to words can be determined:

$$
\begin{gathered}
link\_top_{ww}(1,3)\\
link\_top_{ww}(2,1)\\
link\_top_{ww}(4,1)\\
link\_top_{ww}(3,1)\\
link\_top_{ww}(2,6)\\
link\_top_{ww}(4,6)\\
link\_top_{ww}(3,6)\\
link\_top_{ww}(5,3)\\
link\_top_{ww}(7,3)\\
link\_top_{ww}(6,3)
\end{gathered}
$$

This can be illustrated by:



The only two-way connections are:

$$
\begin{gathered}
thin\_link(1,3)\\
thin\_link(3,6)
\end{gathered}
$$

This, together with the fat link structure, determines the connectivity graph.

$$
\begin{gathered}
linked(2,3)\\
linked(4,3)\\
linked(5,6)\\
linked(7,6)\\
linked(1,3)\\
linked(3,6)
\end{gathered}
$$

We will now describe numerous clauses that describe this model.

5.4 Different link types - encoding

Links between connectors. The set of these variables $link_{cc}((w_i,p_i),(w_j,p_j))$ is completely the same as for sentences that do not contain conjunctions.

Links between connectors and words. If the word w_j is an ordinary word, then the connector (w_i, p_i) is linked to it and $link_{cw}((w_i, p_i), w_j)$ holds iff there is a connector (w_j, p_j) such that $link_{cc}((w_i, p_i), (w_j, p_j))$ holds.

If w_j is a connective word, then (w_i, p_i) can be linked directly to one of its connectors in which case it must not have fat-links down, or if has fat-links down it can be linked to connectors of the words below w_j in the fat-link tree.

Notice that in some cases it can trivially be detected the the link between the connector (w_i, p_i) and the word w_j cannot be established.

1. w_j is an ordinary word and it does not have any connector that matches (w_i, p_i).

2. w_j is a connective word, it does not have any connector that matches (w_i, p_i) and either (w_i, p_i) is not an andable connector, or there is no word w_k such that $\mathcal{B}(w_i, w_k, w_j)$ with a connector that can match (w_i, p_i) or there is no word w_k such that $\mathcal{B}(w_j, w_i, w_k)$ with a connector that can match (w_i, p_i).

In order to reduce the size of the formula and to make the solving process faster, we chose to eliminate those variables from the formula.

If the variable $link_{cw}((w_i, p_i), w_j)$ is not eliminated then one of the two cases does not hold.

- If w_j is an ordinary word, then there exists at least one matching connector (w_j, p_j) and the following condition is generated.

$$link_{cw}((w_i, p_i), w_j) \Leftrightarrow \bigoplus_{p_j} link_{cc}((w_i, p_i), (w_j, p_j))$$

- If w_j is a connective word, then it either has a matching connector (w_j, p_j) or the connector is andable and there are words w_{k_1} such that $\mathcal{B}(w_i, w_{k_1}, w_j)$ and w_{k_2} such that $\mathcal{B}(w_j, w_i, w_{k_2})$ with connectors that can match (w_i, p_i).

  If the word is used as an ordinary word, then one of its matching connectors must be connected (if there are matching connectors at all).

$$w_j \implies link_{cw}((w_i, p_i), w_j) \Leftrightarrow \bigoplus_{p_j} link_{cc}((w_i, p_i), (w_j, p_j))$$

  This condition can be converted to clauses as:

$$w_j \wedge link_{cw}((w_i, p_i), w_j) \Rightarrow \bigvee_{p_j} link_{cc}((w_i, p_i), (w_j, p_j))$$

$$link_{cc}((w_i, p_i), (w_j, p_j)) \Rightarrow link_{cw}((w_i, p_i), w_j), \text{ for all } p_j$$

  Of course, there should also be the exclusive disjunction conditions.

$$\neg link_{cc}((w_i,p_i),(w_j,p'_j)) \ \vee \ \neg link_{cc}((w_i,p_i),(w_j,p''_j)), \quad p'_j < p''_j$$

If the word w_j is used as a special connective word (i.e., it has two fat-links down), but if the connector (w_i,p_i) is not andable or there is no word w_k such that $\mathcal{B}(w_i,w_k,w_j)$ with a connector that can match (w_i,p_i) or there is no word w_k such that $\mathcal{B}(w_j,w_i,w_k)$ with a connector that can match (w_i,p_i), then there can be no connection between (w_i,p_i) and w_j. This is encoded by the clause:

$$w_j^{fl\_d} \ \Rightarrow \ \neg link_{cw}((w_i,p_i),w_j)$$

In other case the connector (w_i,p_i) is an andable connector and there is a nonempty collection of words W_{k_1} such that for each $w_{k_1} \in W_{k_1}$ it holds that $\mathcal{B}(w_i,w_{k_1},w_j)$ and variable $link((w_i,p_i),w_{k_1})$ is not eliminated, and a nonempty collection of words W_{k_2} such that for each $w_{k_2} \in W_{k_2}$ it holds that $\mathcal{B}(w_i,w_j,w_{k_2})$ and variable $link((w_i,p_i),w_{k_2})$ is not eliminated.

In that case, the following must hold:

$$w_j^{fl\_d} \implies$$
$$\left(link_{cw}((w_i,p_i),w_j) \ \Leftrightarrow \ \left(\bigvee_{w_{k_1}\in W_{k_1}} (fat\_link(w_{k_1},w_j) \wedge link_{cw}((w_i,p_i),w_{k_1}))\right) \wedge \left(\bigvee_{w_{k_2}\in W_{k_2}} (fat\_link(w_{k_2},w_j) \wedge link_{cw}((w_i,p_i),w_{k_2}))\right)\right)$$

In order to convert the last formula to CNF, auxiliary variables $link_{cw}((w_i,p_i),w_j)^{k_1}$ and $link_{cw}((w_i,p_i),w_j)^{k_2}$ are introduced.

In that case, the equation reduces to:

$$w_j^{fl\_d} \implies$$
$$\left(link_{cw}((w_i,p_i),w_j) \ \Leftrightarrow \ link_{cw}((w_i,p_i),w_j)^{k_1} \wedge link_{cw}((w_i,p_i),w_j)^{k_2}\right)$$

$$link_{cw}((w_i,p_i),w_j)^{k_1} \ \Leftrightarrow \ \bigvee_{w_{k_1}\in W_{k_1}} (fat\_link(w_{k_1},w_j) \wedge link_{cw}((w_i,p_i),w_{k_1}))$$

$$link_{cw}((w_i,p_i),w_j)^{k_2} \ \Leftrightarrow \ \bigvee_{w_{k_2}\in W_{k_2}} (fat\_link(w_{k_2},w_j) \wedge link_{cw}((w_i,p_i),w_{k_2}))$$

Then, the following clauses are generated:

$$w_j^{fl\_d} \wedge link_{cw}((w_i,p_i),w_j) \Rightarrow link_{cw}((w_i,p_i),w_j)^{k_1}$$
$$w_j^{fl\_d} \wedge link_{cw}((w_i,p_i),w_j) \Rightarrow link_{cw}((w_i,p_i),w_j)^{k_2}$$
$$link_{cw}((w_i,p_i),w_j)^{k_1} \wedge link_{cw}((w_i,p_i),w_j)^{k_2} \Rightarrow link_{cw}((w_i,p_i),w_j)$$
$$link_{cw}((w_i,p_i),w_j)^{k_1} \Rightarrow \bigvee_{w_{k_1} \in W_{k_1}} fat\_link(w_{k_1},w_j)$$
$$link_{cw}((w_i,p_i),w_j)^{k_2} \Rightarrow \bigvee_{w_{k_2} \in W_{k_2}} fat\_link(w_{k_2},w_j)$$
$$link_{cw}((w_i,p_i),w_j)^{k_1} \wedge fat\_link(w_{k_1},w_j) \Rightarrow link_{cw}((w_i,p_i),w_{k_1})$$
$$link_{cw}((w_i,p_i),w_j)^{k_2} \wedge fat\_link(w_{k_2},w_j) \Rightarrow link_{cw}((w_i,p_i),w_{k_2})$$
$$link_{cw}((w_i,p_i),w_{k_1}) \wedge fat\_link(w_{k_1},w_j) \Rightarrow link_{cw}((w_i,p_i),w_j)^{k_1}$$
$$link_{cw}((w_i,p_i),w_{k_2}) \wedge fat\_link(w_{k_2},w_j) \Rightarrow link_{cw}((w_i,p_i),w_j)^{k_2}$$

The last two formulae are families of clauses and there should be one for each $w_{k_1} \in W_{k_1}$ and $w_{k_2} \in W_{k_2}$. The last two types of clauses must hold, because if there is a fat-link from w_{k_1} to w_j, there cannot be any other fat-link from a word in W_{k_1} to w_j.

Although these clauses suffice to have a complete model, the search could be made faster if the following clauses are also generated.

$$link_{cw}((w_i,p_i),w_j)^{k_1} \Rightarrow \bigvee_{w_{k_1} \in W_{k_1}} link_{cw}((w_i,p_i),w_{k_1})$$
$$link_{cw}((w_i,p_i),w_j)^{k_2} \Rightarrow \bigvee_{w_{k_2} \in W_{k_2}} link_{cw}((w_i,p_i),w_{k_2})$$

Implementation:

Each connector is processed and all specified clauses are generated as a part of the method

```
void SATEncoder::generate_satisfaction_for_connector(
      Exp* e, int wi, int& pi,
      char* var_name, int ancestor_cost);
```

For each connector (w_i,p_i) its set of $link_{cc}((w_i,p_i),(w_j,p_j))$ variables is easily determined by consulting the `_word_tag[wi].get_connector(pi).matches`.

The method

```
bool SATEncoder::link_cw_possible(int wi, int pi, int wj);
```

is used for checking if a connector between the connector (w_i,p_i) and the word w_j is possible and whether the variable $link_{cw}((w_i,pi),w_j)$ is eliminated. This method uses the:

```
bool SATEncoder::link_cw_possible_fl(int wi, int pi, int wj);
```

which checks if the connection is possible when w_j is used as a special word and has fat-links down. This method can be also used on its own.

Generating all the complicated clauses for the connective with fat-links down is delegated to the method

```
void SATEncoder::generate_link_cw_connective_definition();
```

Direct links from connectors to words. Recall that the link from a connector to (w_i, p_i) to the word w_j is called direct if there is a link between the connector (w_i, p_i) and the word w_j and that link it is not inherited from above, i.e., if there is no connective word w_k such that the w_j is fat-linked up w_k, such that the connector (w_i, p_i) is linked to w_k. Direct links from connectors to words are encoded with $link\_top_{cw}((w_i, p_i), w_j)$ variables.

- If the connector (w_i, p_i) is not an andable connector, then the direct link is established iff an ordinary link is established. This is described by the relation:

$$link\_top_{cw}((w_i, p_i), w_j) \Leftrightarrow link_{cw}((w_i, p_i), w_j)$$

- If the connector (w_i, p_i) is an andable connector, it is directly connected to w_j if there is no connective word w_k such that w_j is fat-linked up to w_k and that the connector (w_i, p_i) is connected to it. Let W_k be the set of connective words w_k such that w_k and w_j are on the same side of w_i (i.e., $\neg\mathcal{B}(w_j, w_i, w_k)$) and that the variable $link_{cw}((w_i, p_i), w_k)$ is not eliminated. Then it holds that

$$link\_top_{cw}((w_i, p_i), w_j) \iff link_{cw}((w_i, p_i), w_j) \wedge \neg\left(\bigvee_{k \in W_k} fat\_link(w_j, w_k) \wedge link_{cw}((w_i, p_i), w_k)\right)$$

This formula can be converted to clauses as:

$$link\_top_{cw}((w_i, p_i), w_j) \Rightarrow link_{cw}((w_i, p_i), w_j)$$

$$link\_top_{cw}((w_i, p_i), w_j) \Rightarrow \neg fat\_link(w_j, w_k) \vee \neg link_{cw}((w_i, p_i), w_k)$$

$$\neg link\_top_{cw}((w_i, p_i), w_j) \wedge link_{cw}((w_i, p_i), w_j) \Rightarrow \bigvee_{k \in W_k} fat\_link(w_j, w_k)$$

$$\neg link\_top_{cw}((w_i, p_i), w_j) \wedge link_{cw}((w_i, p_i), w_j) \wedge fat\_link(w_j, w_k) \Rightarrow link_{cw}((w_i, p_i), w_k)$$

Second and fourth formulae are families of clauses and there should be one of these for each $k \in W_k$. The clauses of the fourth family must hold, because if there is a fat-link up from w_j to w_k, then there can be no other fat-link up from w_j.

Although these clauses suffice for the completeness of the encoding, the following clauses could be added to speed up the search process:

$$\neg link\_top_{cw}((w_i, p_i), w_j) \wedge link_{cw}((w_i, p_i), w_j) \Rightarrow \bigvee_{k \in W_k} link_{cw}((w_i, p_i), w_k)$$

Implementation:

Defining $link\_top_{cw}((w_i, p_i), w_j)$ variables also happens as a part of the method

```
void SATEncoder::generate_satisfaction_for_connector(
      Exp* e, int wi, int& pi,
      char* var_name, int ancestor_cost);
```

Generating clauses is delegated to methods

```
void generate_link_top_cw_iff_link_cw();
void generate_link_top_cw_definition();
```

The set Wk is determined using the method `link_cw_possible`.

Direct links from words to words.

- If w_i is an ordinary word, then there is direct link from it to the top of the word w_j iff there is a connector on w_i which is linked to top of w_j.

$$link\_top_{ww}(w_i, w_j) \Leftrightarrow \bigvee_{(w_i, p_i)} link\_top_{cw}((w_i, p_i), w_j)$$

- If w_i is a connective word, but acts as an ordinary one, then the last relation conditionally holds

$$w_i \Longrightarrow link\_top_{ww}(w_i, w_j) \Leftrightarrow \bigvee_{(w_i, p_i)} link\_top_{cw}((w_i, p_i), w_j)$$

If w_i acts as a special word (i.e., it has fat-links down), then it is dirrectly connected to w_j (i.e., connected to its top) iff the two words that are fat-linked up to it are directly connected w_j. These conditions are converted to clauses:

$$link\_top_{ww}(w_i, w_j) \wedge fat\_link(w_k, w_i) \Rightarrow link\_top_{ww}(w_k, w_j), \quad \mathcal{B}(w_i, w_k, w_j) \vee \mathcal{B}(w_k, w_i, w_j)$$

$$fat\_link(w_{k_1}, w_i) \wedge link\_top_{ww}(w_{k_1}, w_j) \wedge fat\_link(w_{k_2}, w_i) \wedge link\_top_{ww}(w_{k_2}, w_j) \Rightarrow link\_top_{ww}(w_i, w_j) \quad \mathcal{B}(w_i, w_{k_1}, w_j) \wedge \mathcal{B}(w_{k_2}, w_i, w_j)$$

Thin-links Variables $thin\_link(w_i, w_j)$, $w_i < w_j$ determine if a thin-link has been established between two words. Two words are connected by a thin-link iff there is a direct connection from one to another and vice-versa.

$$thin\_link(w_i, w_j) \Leftrightarrow link\_top_{ww}(w_i, w_j) \wedge link\_top_{ww}(w_j, w_i)$$

Linked words Now all neccessary auxiliary variables have been defined, we can proceed and define $linked(w_i, w_j)$ that determine if the two words are linked and that are crucial for formulating and checking planarity and connectivity constraints. Two ordinary words are linked, iff there is a thin-link between them.

$$linked(w_i, w_j) \Leftrightarrow thin\_link(w_i, w_j)$$

If w_i is a connective word (the case of w_j being a connective word is analogous), then the words are connected iff there is either a thin-link between them or a fat-link from w_j to w_i.

$$linked(w_i, w_j) \Leftrightarrow thin\_link(w_i, w_j) \vee fat\_link(w_j, w_i)$$

If both words are connective words, then they are linked if there is a thin-link between them or a fat-link in either direction.

$$linked(w_i, w_j) \Leftrightarrow thin\_link(w_i, w_j) \vee fat\_link(w_j, w_i) \vee fat\_link(w_i, w_j)$$

6 Guiding

Although a sentence might have a large number of syntactically correct linkages, there is usually just a small number (often that is only one) semantically correct linkages. That is why having a mechanism of ranking different parses which would prioritize semantically correct linkages is very important. The classic link-parser implementation constructs all syntactically correct linkages and then sorts them descending according to a ranking which uses costs (penalties) from the dictionary and the total length of links present in the linkage. The rationale for the latter criterion is that usually only words that are very close should be linked and the total sum of link-lengths should be small.

One of the design goals for the SAT based link-parser implementation was to avoid the need for constructing all syntactically valid linkages of a sentence but to use only top N linkages. In order to have this possible there has to be some search guiding that would increase the probability that the semantically valid linkages are among those top N. This opens up several problems.

First, SAT problem, as originally formulated is the problem of deciding whether there is a satisfying valuation (i.e., a model) for a propositional formula. If it is formulated like this, it is a *decision problem*. Usages of SAT in software industry and in theorem proving community (usually for software and hardware verification) very often need to show that there is no satisfying valuation for a formula. When treated as a decision problem, all satisfying valuations are treated in an equivalent fashion. Heuristic components of SAT solvers, primarily the *decision (i.e. variable selection) heuristic*, traverse the search space so that parts of the search space that contain many potential models are examined with greater priority. Stumbling across a model (what ever model that might be) gives answer to the decision problem.

On the other hand, link-parser application requires SAT to behave very different. Link-parsing can be formulated as an *optimization problem*. Unlike in verification and theorem proving, it is expected that the formula that is being solved has a satisfying valuation, and in most cases even that it has many satisfying valuations. Finding any of them would give the answer to the decision problem, but that answer is usually already anticipated to be positive. The optimization problem that needs to be solved assumes that some ordering of models of the formula is established and it is required that SAT solver finds exactly the model that is the top one in that ordering. This task is much

harder then simply finding any model. As a very unpleasant consequence, state-of-the-art decision heuristics cannot be used without making some smart adjustments. Decision heuristics have to be adjusted in a way which would enforce a prioritized systematic traversal of the search space. In this scenario, priority must be given to parts of the search space that potentially contain the top models, and not to parts of the search space that are filled with models that are easily found but are not among the top in the model ordering.

The way in which a model ordering is constructed in the SAT based link-parser implementation is assign two different numbers to each variable in the formula:

1. *Decision priority*

2. *Decision polarity*

The systematic search builds models so that variables with higher priorities are chosen first and they are assigned to have their preferred polarity. Lets say that the variable v has been assigned its preferred polarity p. Its polarity is going to be flipped to $\neg p$ only after all variables having smaller priority than v have been tested and it is determined that there can no model that contains v in polarity p. There are several drawbacks to this approach.

1. If there is no satisfying valuation which contains v in the polarity p, but the variable v has a high priority, it can take a very large amount of time to investigate all possible valuations of variables with smaller priority than v, because there can be a very large number of such variables. Therefore, it is crucial for a guiding scheme to ensure that when variables are assigned high priorities it is the case that in valid linkages they really occur only in their preferred priorities. If the preferred polarity of a variable can not be determined with a high reliability degree, it must not be given a high decision priority.

2. This total order of variables based on their priorities enforces that measure of quality of parses (i.e., the ordering of parses based on that measure) is calculated as a function of priority of variable that has the opposite priority then its preferred priority. Let us illustrate this on an example. Assume that variable x has priority 3, y has 2 and z has 1 and all variables have the preferred polarity $\top$. In this case, the model $x = \top, y = \bot, z = \bot$ has the better quality then the model $x = \bot, y = \top, z = \top$, because the maximal priority variable x has its preferred value. The values of all other variables are irrelevant for the measure of model quality. It would be natural to have an ordering which would say that the second model $x = \bot, y = \top, z = \top$ has better quality then the first model $x = \top, y = \bot, z = \bot$, because it has two variables set on their preferred polarities and only one that is not. Also, it would be natural to say that two models are of equal quality because the sum of priorities of the variables with missed priority is 3 in both cases. However, although the two latter measures look appealing, only the first one can easily be implemented by using the described technique. It would be hard (although not impossible) to simulate counting and summation in SAT and the two latter orderings would require to have this.

Implementation:

The guiding scheme for the SAT search has to specify priorities and polarities for every variable of the SAT encoding. Since there can be several different guiding schemes, it has been decided to create an API which would allow users to implement different guiding schemes relatively independent from the rest of the system.

Part II

Implementation

7 Mapping between variables and numbers

All propositional clauses consist of variables which are represented by integers. Various SAT encodings introduce a number of different variables. When implementing a SAT encoding there is the need to assign different numbers to different variables. Variables usually have some meaning and are named according to their type and their parameters. For example, variable $link_{cc}((w_i, p_i), (w_j, p_j))$ has the meaning of a link between the connector C_i^+ in the word w_i on the position p_i and the connector C_j^- in the word w_j on the position p_j. On the other hand, when implemented, clauses can only contain a number that is a placeholder for this specific variable. Therefore, a two-way mapping between these higher-level readable descriptions of variables and numbers that represent their internal codes had to be established. The implementation of this functionality is done through the `class Variables`. First the implementation of a mapping that assigns numbers to high-level descriptions of a variable will be described, and after that the inverse mapping which gives higher-level description and information about the variable with a given number will be described.

7.1 Variables to numbers: string to int mapping approach

The easiest way of implementing this functionality could be achieved if higher-level variable representations are kept as strings, and if these strings are mapped to numbers. This functionality is achieved through the function:

```
int Variables::get_string_variable(char* name);
```

If there is already a number assigned to this name, it is returned. If there is not, then a fresh number is assigned to this name and it is memorized and returned. This interface is implemented through a mapping between names and numbers which is stored in a *trie* (short for re*trie*val) data-structure implemented in the `class Trie`. Decision to use a trie was made after it has been noted that many variables introduced by a specific encoding for the link-grammar word-tags that uses Tseitin encoding described in the section 2.3 have very similar names and that it is often the case that one variable name is a prefix of several other names. In order to save some space, the trie is implemented so that it only supports variables names built-on a small alphabet Σ.

Although the described string-to-int mapping approach is very uniform, careful profiling showed that it can be sometimes very slow. The critical operation turned out to be the creation of strings from data that represent variables. For the $link_{cc}((w_i, p_i), (w_j, p_j))$ variables described above, simple `sprintf(name, "link_%d_%d_%d_%d", wi, pi, wj, pj)` turned out to be extremely slow – much slower than the retrieval from the trie itself. In order to improve efficiency, faster implementations of printing strings and integers into a character buffer are implemented (`fast_sprintf`). Although this helped to some extent, it was decided to introduce other kinds of mappings for specific kinds of variables.

7.2 Variables to numbers: int tupples to int mapping approach

Using this approach a separate registry for several different types of variables is kept. The public interface of the `class Variables` is extended by functions such as[1]:

```
int Variables::get_linked_variable (wi, wj);
int Variables::get_link_variable   (wi, pi, wj, pj);
...
```

The implementation of these functions is made in a way which is optimal for these specific kinds of variables.

For example, if we consider a $linked(w_i, w_j)$ variables which describes the fact that there is a link between the words w_i and w_j, it can be noticed that it is uniquely determined by its (w_i, w_j) pair. Therefore, the fastest retrieval of variable numbers for these variables can be achieved if a map from pairs of ints to ints is kept. Of course, it is best to use a matrix of ints (the matrix datatype is implemented in `Matrix<T>` and for symmetric matrices in its subclass `MatrixUpperTriagle<T>`).

Similarly, for the $link_{cc}((w_i, p_i), (w_j, p_j))$ variable already described in a previous example, it can be noted that this specific *link* variable is uniquely determined by its (w_i, p_i, w_j, p_j) variable. The mapping from this 4-tupples of ints to ints is implemented by keeping a matrix of maps from int pairs to ints. The matrix is indexed by the (w_i, w_j) pairs, because they come from a range known in advance (that is $[0, n)$, where n is the length of the sentence) and it is expected that for this matrix to be quite dense. On the other hand, mappings from (p_i, p_j) to ints are stored in `std::map< std::pair<int, int>, int >`, because the range of positions p_i is not known in advance and this mapping is expected to be quite sparse.

7.3 Numbers to variables

Once the propositional model of a formula is found, it consists of integers representing variable numbers. In order to reconstruct a linkage from these numbers a mapping inverse to mappings described previously should be also

[1] Note that various encodings introduce much more variable types and these two variable types shown in this document are just representatives of the technique used for other variable types as well.

implemented. First, it can be noted that only some variables are relevant for reconstructing the linkage from the propositional model and these inverse mappings are kept only for those specific kinds of variables. The first question that needs to be answered is: "What are the numbers of all variables of this specific type?". In order to implement answers to those questions the interface is extended by methods:

```
const std::vector<int>& Variables::get_linked_variables() const;
const std::vector<int>& Variables::get_link_variables() const;
...
```

These functions are trivially implemented by maintaining int vectors for relevant variable types.

The next question that needs to be addressed is: "What variable does the number v represent? Give me information about this variable". The first approach was to return the information in a form of string (e.g., `"link_3_2_Sp_5_6_Spx"`), but in order to retrieve the information from the returned string you need to split it and parse it which is slow and cumbersome. Therefore, for variables that are of interest for model reconstruction, structures that contain additional information about those variables are defined and kept. We give one example of such functionality.

```
struct Variables::LinkVariable {
  int left_word;
  int left_position;
  const char* left_connector;
  int right_word;
  int right_position;
  const char* right_connector;
};

const Variables::LinkVariable&
            Variables::get_link_variable(int variable_number);
```

Structures defining information about all variables are memorized during variable construction (i.e., when they are assigned a number). Although one may be concerned about the memory consumption because all this bookkeeping, it turns out that this is quite neglectable when compared by the size of the CNF formula that is maintained and which constantly grows during the SAT solving process.

Since the classic link parser implementation is done in C, not in C++, and the `class Variables` is just a simple technical class we decided not to use the advanced OO concepts such as inheritance or polymorphism in its implementation.

For the list of all variable types currently supported by the class variables, the reader should consult the code itself and the comments listed in the code.

8 CNF conversion routines

The main task of the SAT encoding phase is to generate the formula that describes all link-parsing conditions in conjunctive normal form and to pass all its clauses to the SAT solver. Since the SAT solver used is MiniSAT, the clauses are expected to be prepared using its internal data structures (`vec` - for vectors, and `Lit` for literals) instead of standard ones. The main burden of creating the formula lies on the `class SATEncoder`. It builds clauses and passes them to MiniSAT using the method:

```
void add_clause(vec<Lit>& clause);
```

An convenient method for asserting facts is the method

```
void add_literal(Lit l);
```

which builds a single literal clause and passes it to MiniSAT.

Many link-grammar constraints are usually described as equivalences of certain type. This is why, it has been decided to implement special functions that preform conversion to CNF of certain types of formulae. All these functions are implemented as methods of the `class SATEncoder`. As these clauses are directly passed to the SAT solver MiniSAT, in order to obtain maximal efficiency, these CNF conversion routines also expect their parameters in MiniSAT data structures. Now we give a short summary of these CNF conversion methods.

$\vee$ **definitions.** Formulae that represent definition of a literal by a classic disjunction of several other literals:

$$l \Leftrightarrow l_1 \vee \ldots \vee l_k$$

is converted to CNF using the method:

```
void generate_classic_or_definition(Lit lhs, vec<Lit>& rhs);
```

$\wedge$ **definitions.** Formulae that represent definition of a literal by a classic conjunction of several other literals:

$$l \Leftrightarrow l_1 \wedge \ldots \wedge l_k$$

is converted to CNF using the method:

```
void generate_classic_and_definition(Lit lhs, vec<Lit>& rhs);
```

$\oplus$ definitions. Formulae that represent definition of a literal by an exclusive disjunction of several other literals:

$$l \Leftrightarrow l_1 \oplus \ldots \oplus l_k$$

are converted to CNF using the method:

```
void generate_xor_definition(Lit lhs, vec<Lit>& rhs);
```

This method uses the `generate_or_definition` along with the

```
void generate_xor_conditions(vec<Lit>& lits);
```

The last method generates pair-wise disjunctions of literals and therefore generates a quadratic number of clauses. It can be also used on its own.

`&` definitions. Formulae that represent definition of a literal by a strong conjunction of several other literals:

$$l \Leftrightarrow l_1 \,\&\, \ldots \,\&\, l_k$$

are converted to CNF using the method:

```
void generate_and_definition(Lit lhs, vec<Lit>& rhs);
```

Conditional definitions. It has been noted that in some cases equivalencies conditionally hold, i.e., they are implied by a certain condition. For example,

$$c \Longrightarrow (l \Leftrightarrow l_1 \,\mathtt{op}\, \ldots \,\mathtt{op}\, l_k)$$

where `op` is some connective ($\vee$, $\wedge$, $\oplus$, `&`). These are converted to clausal form using several methods of the form:

```
void generate_conditional_xxx_definition(
         Lit condition, Lit lhs, vec<Lit>& rhs);
```

9 Representing word-tags

9.1 Word-tag representation in the classical link-parser implementation

Word-tags for all words are kept in the dictionary. When the dictionary is parsed word-tags are represented by the n-ary tree structure called `Exp` and defined in `structures.h`. The sentence data structure `Sent` contains, for each word, a list `Sent::x` of expressions from the dictionary. If there are several dictionary entries for a single word, they are kept in several different trees in the list `Sent::x`.

9.2 Basic simplification of word-tags.

When the sentence is created, word-tags can be simplified by eliminating connectors that cannot connect to any other connector in the sentence. This operation is implemented in the classical link-parser implementation by the function `void expression_prune(Sentence sent)` implemented in `prune.c`. The SAT implementation uses this functionality and calls this function before the parsing begins.

9.3 Caching information from the word-tags

In order to have effective encoder implementation, once the word-tag expressions are simplified, some information that they contain are converted to more convenient format and cached for faster further reference. This cached representation of word-tags is kept in `class WordTag`. Basically, this class contains information about all connectors contained in a word-tag expression. If a word contains several different expressions they are merged into a single one before they are cached. This operation will be more thoroughly discussed later. When this is done, each connector in the sentence is uniquely determined by the word that it is on (i.e., its word-tag), and its position in the simplified and merged `Exp` tree. These positions correspond to the preorder (dfs) traversal of the tree. Since we tried not to alter any existing code, but only to add the new

code, a new structure that extends the Connector data-structure defined in structures.h is implemented.

```
struct PositionConnector {
  Connector* connector;
  int word;
  int position;
  std::vector<PositionConnector*> matches;
  ...
};
```

This structure, along with the classic information about the connector (contained in Connector data-structure) also keeps track of the word that this connector is on and its position in the word-tag (i.e., in its expression tree). Also, what is most important, it contains a collection of all other connectors in this sentence that this connector could connect to.

The class WordTag represents the word-tag of a single word in a sentence and it keeps track of all its left-pointing and right-pointing connectors. This information is available through the following methods:

```
const std::vector<PositionConnector>&
             WordTag::get_left_connectors();
const std::vector<PositionConnector>&
             WordTag::get_right_connectors();
```

It also offers the functionality of getting the information about its specific connectors.

```
int WordTag::num_connectors();
const PositionConnector& WordTag::get_connector(int position);
```

It can be also very quickly checked if this word-tag can match a given connector (again uniquely determined by its word and position).

```
bool WordTag::can_match(int wi, int pi);
```

WordTag objects are build by the SATEncoder once before the parsing starts and they are frequently used by the encoder during the encoding process while building the CNF formula.

Note: the notion of whether two connectors match changes depending whether the sentence contains conjunctions or it does not. The classical link-parser implementation offers the function `int match(Connector* a, Connector* b, int aw, int bw)` which should be used in cases when the sentence does not contain conjunctions and the function `int prune_match(int dist, Connector* a, Connector * b)` which should be used in cases when the sentence contains conjunctions. The reason this is rather subtle: in conjunctive sentences like "the cat and dog run", singular Ss^+ connector on "cat" and "dog" should match plural Sp^-, while in ordinary sentences "the cat run" it should not. The class WordTag has to get the reference to the sentence itself, and then it automatically determines which matching function to use based on the structure of the sentence. The two matching functions require that internal data structures of the sentence Sent are computed and this is done once before the building of the word-tags by calling:

```
build_deletable(sent, 0);
build_effective_dist(sent, 0);
count_set_effective_distance(sent);
```

10 MiniSAT modifications

In order to use it for the link parsing, the solver MiniSAT had to be adapted in several ways.

10.1 Adding clauses "online"

As SAT is primarily a decision problem, SAT solvers usually stop the search process after they find the first model of the formula. Link-parsing application requires that more than one valid linkage is constructed. Therefore, we had to make modifications that would allow MiniSAT to continue the search even after a model has been found and to somehow force it to find models different than the one that has been found. The simplest way to forbid a model is to add a single clause that would forbid it. Namely, if model is seen as a conjunction of literals

$$l_1 \wedge l_2 \wedge \ldots \wedge l_k,$$

then its negation

$$\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_k$$

is a clause that is false in this model and so it forbids it. This clause can be added to the formula and the search can be restarted. After a model is found, MiniSAT automatically resets its state to the initial state in which the assertion trail is empty. This operation discards a lot of work that has been done and is implicitly stored in the assertion trail data-structure. That is why we have decided to prevent this. In a modified version of the solver, when the model is found, the trail remains intact. But this raises another problem. The method `addClause` of the MiniSAT solver used for adding clauses expects that the solver is initial state and that there are not decision literals present on the assertion trail. In order to make possible to add clauses "online" i.e. to add them during the search, when the solver is not in its initial state, we had to implement a new method

```
void Solver::addClauseOnline(vec<Lit>& clause);
```

This method uses a non-trivial technique that analyzes if the clause that is being added is conflict, unit or just an ordinary clause with respect to the

current solver state, and adjusts the state according to the clause that is being added. If this clause is a conflicting clause (which is most frequently the case in the scenario where model forbiding clauses are being added), then the standard conflict analysis procedure is performed and the assertion trail is backtracked so that the conflict is resolved. Special attention had to be put on the watch literals of the clause.

10.2 Adding binary clauses

A large percent of the clauses in the inital encoding are binary clauses (i.e., clauses that contain exactly two literals). The MiniSAT method `addClause` does not treat binary clauses different from other clauses. In order to eliminate duplicate literals, it calls the method `sort`. Profiling showed that this method call takes significant amount of time, and can be omitted for binary clauses since duplicates can be detected without sorting with just a single comparison. This cheap tricks reduces the encoding time by few percent.

10.3 Decision strategy

In order to enable guided search, decision strategy (i.e., variable and polarity selection strategies) had to be altered. This modification did not require much coding, but is crucial for solver efficiency. MiniSAT maintains heap containing variables sorted descending by their decision priority. Decision priorities directly correspond to so called variable activities which are constantly and dynamically updated during the search process. When a decision needs to be made, the top variable of the heap (the one with the highest priority i.e., activity at that point) is selected. Depending on the polarity selection strategy, this variable is assigned always negative, always positive or always random polarity. In order to implement static decision priorities for variable that guide the SAT we used the built-in heap and variable activity vector. The code was extended so that these numbers can be set before the search starts. The lines of code that dynamically update variable activities are excised so that all variables have their initial decision priorities during the search. Polarity selection strategy is done using the preferred polarity that is also specified for each variable before the search starts. Priorities can be arbitrary floating point numbers and their order is only thing that matters. Polarities must be numbers from $[0, 1]$. Polarity is selected using the random number generator which generates a random number from $[0, 1)$. If that number exceeds the given polarity number the polarity is set to true, and otherwise it is set to false.

Part III

Evaluation

11 Results

First we give run-time results for the encoding that does not support conjunctive sentences.

Sentence no.	Encoding	First model	Relevant model	All models
1	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00
3	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00

12
Conclusions and further work

Project outcome. The main outcome of this project is that a *fully functional link-parser based on SAT solving* has been developed and implemented. Both the theoretical foundation and an implementation alternative to the classic one have been developed. That would not been possible if a large amount of the classic link-parser codebase had not been used. However, the central search component of the parser is written from scratch and it completely relies on the SAT solving technique.

Complexity of link-grammars. The classic link-parser implementation is a non-trivial piece of software that has been developed for many years. Although the paper [1] is a great introduction into the SAT encoding of link-grammars, it contained some minor errors that had to be corrected in order to get a valid encoding. Also, the link-grammar formalism is much more complicated then it has been anticipated in this paper and the project proposal. Some of the important link-grammar features that have not been mentioned in [1] are for example:

- dealing with conjunctive sentences,
- post-processing,
- arbitrary boolean structure of the word-tags,
- unusual semantics of link-grammar connectives `&` and `or`,
- connector matching rules – unification of connectors and forming of connector labels,
- different pruning techniques – power pruning, pp pruning etc.
- etc.

In order to get a fully functional parser, during the project duration all these features had to be invented and implemented. Some of these show to be easy to solve, but some (e.g., conjunctive sentences) showed to be extremely hard and required complicated algorithms and much work and time.

Comparison of DNF and CNF approach. Carefull examination of classic link-parser implementation shows that it uses sophisticated DNF based algorithms. On the other hand, SAT based link-parser is uses CNF based algorithms.

DNF based algorithms used in classic link-parser imlementation suffer from the problem that the number of disjuncts can become unmanageably large for sentences that are long and complicated. The time and memory required for generating and storing the DNF formula becomes inacceptably large. CNF based approaches do not suffer from these problems. Even for very long and complicated sentences, formulas remain manegeable and are generated reasonably fast. On the other hand, once the DNF formula is constructed classic implementation can perfrom the search for satisfying solutions very fast. This indicates that a breaktrough with the SAT approach can happen with long and complicated sentences which are the bottleneck of the classic implementation. However, the search time for the CNF formula grows with the formula and can be very long for complicated formulae.

Problem structure. One of the main drawbacks of the proposed SAT approach is that it does not use the information about the problem structure. Namely, planarity conditions imply that whenever a link between two non-neighboring words w_i and w_j is established, the problem is divided in two completely independent components. The "outter" component consists of words between the left-wall and w_i and words between w_j and the right-wall, and the "inner" component consists only from words that are between w_i and w_j. Establishing links in one of these two components cannot have any effect on the other one. Classic DNF based link-parser implementation uses this information and whenever it establishes a link, it examines the inner component exhaustivelly (assuming it is smaller then the outter) and constructs all solutions for the inner component. Since the solutions of the inner component depend only on a single link between words w_i and w_j that has been established in the beggining, the classic implementation uses the memoization technique and keeps track of the whole solution set for the inner component. Whenever the same link between w_i and w_j is established in the future, the memorized solution set is returned and there is no need for solving the same inner component again.

The SAT solver does not have this global information about the problem structure available, and it does not use the fact that a link splits the search space to two independent components. The SAT search continues to jump between the two components and usually spends a lot of time solving the larger, outter component. It usually turns out that the inner component is unsatisfiable, but this is detected quite late, when a lot of time has been wasted in examination of the outter component. Instead of memoization, SAT solvers use the learning which, as a general technique, cannot compete with the memoization that takes into account the specific structure of the problem.

Performance analysis. Underlying algorithms of the classic link-parser implementation are used in a way which both enables and requires enumerating all valid parses of a sentence. Only when all linkages are constructed, they are sorted according to some parse rankig order and best linkages are reported. General purpose SAT solvers cannot outperform these specific sophisticated

algorithms that take into account the specific problem structure when it comes to enumerating all syntactically valid linkages of a sentence. However, since formula generating time is usually signifficantly smaller in the CNF approach, the first several linkages of the sentence can be found significantly faster then in the DNF based, classic approach.

Having in mind the unexpected complexity and plethora of features of the link-grammar formalism, it has been a very hard task to develop an implementation during the limited project duration period that would significantly outperform the classic implementation. However, the results given clearly show that the SAT based parser that has been developed can in many cases find several valid linkages of a sentence in significantly less amount of time than the classic implementation does.

In the authors opinion, there is still much space for improving performance. This should be done in several major directions: (i) exploiting the specific problem structure during the SAT search (ii) implementing good search guiding schemes (iii) improving the SAT encoding of conjunctive sentences.

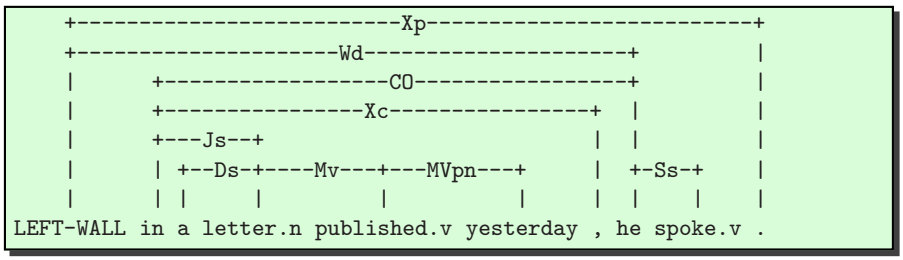
Guiding. In order to guarantee that the semantically valid linkages are among the first several found, an advance semantic/statistic search guiding scheme has to be developed. The SAT based parser that has been developed offers its developers an API that enables them to implement and experiment with different guiding schemes, without the need to know all the SAT encoding details and details about the SAT implementation. During the project, several prototype guiding schemes are implemented. In order to further improve efficency of the search more advance guiding schemes must be developed. This is left for further work, because of the duration period of the project.

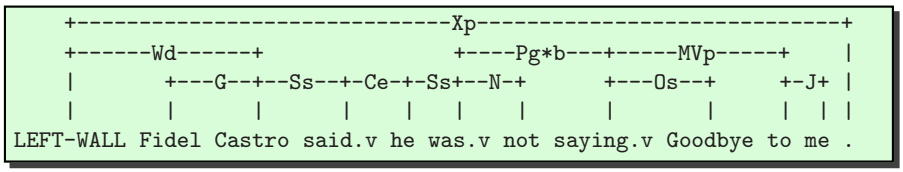
It has been observed that guiding heuristics has much effect on the efficiency of the search. Experiments show that if the guiding heuristic makes a single wrong decision early in the begining of the search process, this can seriously downgrade the performance. Although better guiding heuristic can minimize this effect, the improvement of the overall guiding strategy and making it more robust would be quite beneficial and neccessary for the future developement.

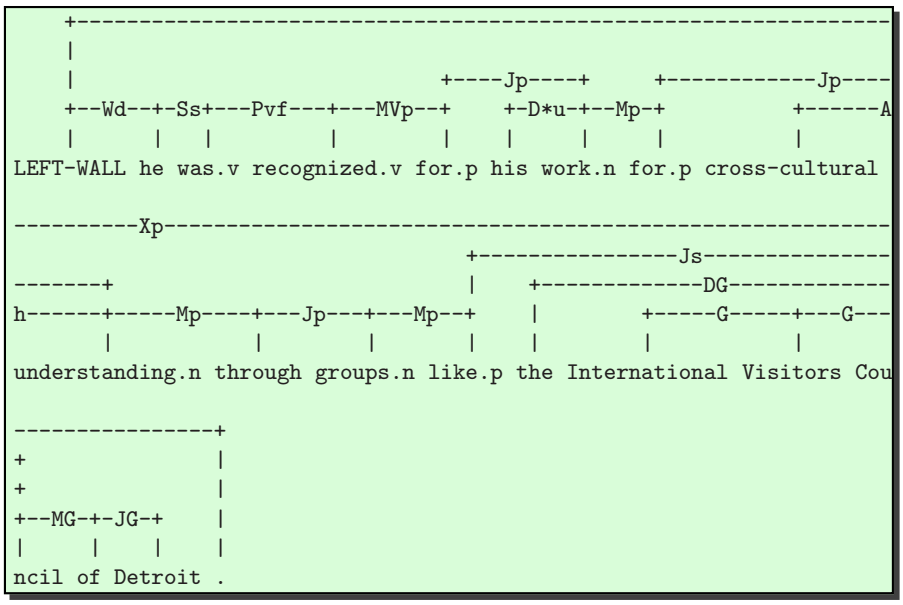
Conjunctive sentences. The toughest part to implement was the support of conjunctive sentences by using the *fat-link* technique. The classic link-parser implementation also pays very much attention to conjunctive sentences. Conjunctive sentence handling represents a very large percentage of original implementation code. Experiments with the classic link-parser implementation show that long conjunctive sentences are critical for its performance, and that classic link parser implementation fails to solve many of them in a reasonable amount of time. Although this problem showed up to be crucial, it has not been anticipated when the project proposal was written and the project proposal does not mention it. Since the conjunctive handling mechanism is the weakest feature of the classic link-parsing implementation, it has been decided to spend almost one half of the project period to implement conjuncting handling in the SAT based parser. This has been done and the encoding of conjunctive sentences has been completely implemented what makes the SAT based parser fully functional.

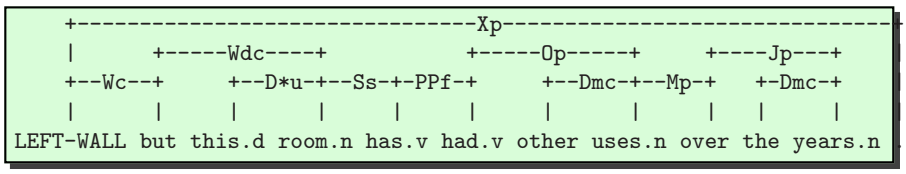
Final conclusion. The developed SAT based link parser implementation is fully functional and it often outperforms the classic link-parser implementation when it comes to enumerating just first several syntactically valid linkages according to some search guiding criterion. Even with using the prototype guiding schemes, the semantically correct parses are usually among the first ones found. Still, it cannot yet be said that it *"significantly (by an order of mangnitude) outperforms"* the classic implementation as it has been hypothesized. It is the authors opinion that this result could potentially be achieved if yet more work is invested in developing guiding schemes and impoving the SAT encoding, especially for the conjunctive sentences, but this still remains just a hypothesis.

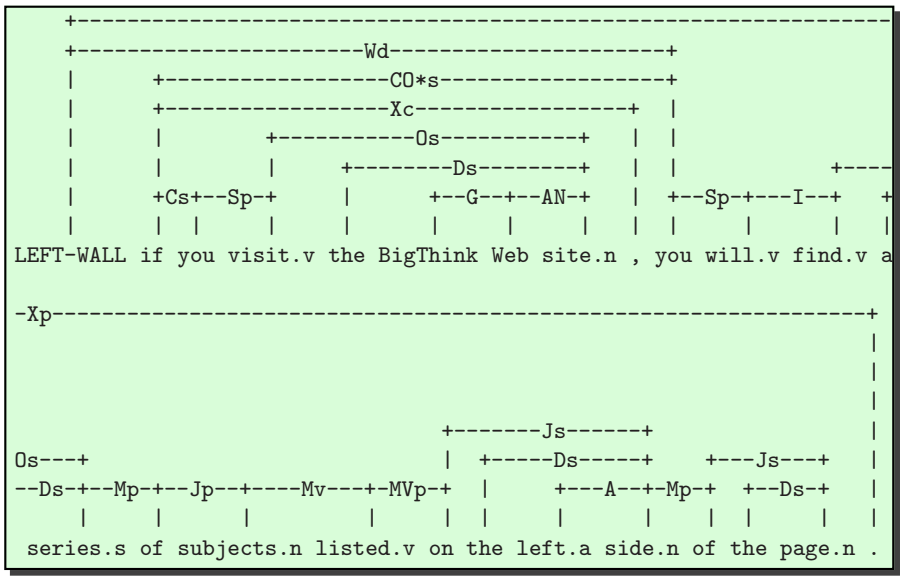
A
Sentences


LEFT-WALL in a letter.n published.v yesterday , he spoke.v .


LEFT-WALL Fidel Castro said.v he was.v not saying.v Goodbye to me .


LEFT-WALL he was.v recognized.v for.p his work.n for.p cross-cultural
understanding.n through groups.n like.p the International Visitors Cou
ncil of Detroit .


LEFT-WALL but this.d room.n has.v had.v other uses.n over the years.n


LEFT-WALL if you visit.v the BigThink Web site.n , you will.v find.v a
-Xp
Os---+
--Ds-+--Mp-+--Jp--+----Mv---+-MVp-+
series.s of subjects.n listed.v on the left.a side.n of the page.n .

Bibliography

[1] P. Janičić, B. Goertzel, *Parsing Based on Link-Grammars and SAT Solvers*, unpublished draft paper.