Syntactic Dependency Graph: a Compact Representation of Alternative Dependency Structures

Tomasz Obrębski[1]

Abstract

The paper presents a parsing technique based on the concept of a syntactic dependency graph. The graph is used to represent alternative dependency structures. Its nodes represent possible morphological interpretations of words and arcs represent possible dependencies between them. The dependency graph replaces the parse chart. We show how to build the graph, find possible root nodes, and construct possible dependency trees. During the graph construction no computation is repeated, no information is duplicated, and all partial solutions are proposed exactly once, though three kinds of ambiguities may be present at the same time. The algorithms are efficient, however only simple and relatively weak grammars may by handled within this approach.

Keywords: parsing, dependency grammar, dependency graph

1 Introduction

The paper presents a parsing technique based on the concept of *a syntactic dependency graph* (*dependency graph* in short). A dependency graph serves as a compact representation of alternative dependency structures ([3], [5]) assigned to a sentence. The nodes of this graph represent possible morphological interpretations of words and the arcs represent possible dependencies between them. We only consider projective dependency structures.

The key property of dependency parsing is that all nodes of a dependency structure being built are known a priori (in opposition to phrase structure parsing, where only leaves are given). Instead of considering alternative subtrees, which is a common practice in tabular parsing (cf [1], [2] for dependency parsing examples), we only consider alternative arcs. All computations are carried on a fixed set of nodes without the necessity of duplicating any of them.

The dependency graph replaces the parse chart. During the graph construction no computation is repeated, no information is duplicated and all partial solutions are proposed exactly once, though three kinds of ambiguities may be present at the same

[1] the Technical University of Poznań, Poland, e-mail: obrebski@ai-kari.put.poznan.pl
This research was partially supported by the Technical University of Poznań, Project DS-45-295/200.

time. In a uniform manner we deal with ambiguities in sentence segmentation, syntactic category assignment, and syntactic structure description. The use of a dependency graph imposes, however, several important restrictions: only binary relations between words (head and dependent) can be considered, attribute values may not be propagated throughout the syntactic structure, attributes may not be assigned to substructures, only to individual nodes or arcs.

2 Morphological graph

The ambiguous output of morphological analysis is represented here as a directed acyclic graph (similar to [4]), whose nodes correspond to possible morphological interpretations of the words of a sentence and arcs represent the direct succession relation between them. Each node is assigned a syntactic category. In this way we can represent ambiguities arising in assigning syntactic categories to words as well as in sentence segmentation (cf the example below). Each path of the graph, leading from one of the *initial nodes* (without a predecessor) to one of the *final nodes* (without a successor), is a possible unambiguous morphological interpretation of the sentence. Such a path will be called *a full path* (in opposition to *a partial path*). The graph will be called *a morphological graph*.

Formally, a morphological graph is a triple $< W, \prec, c >$, where W is a nonempty set (each member represents one potential morphological interpretation of a word), $\prec \subset W \times W$ is a direct succession relation (besides the relation $\prec$ we will also use the precedence relation $<$ where convenient), and $c : W \rightarrow K$ is a function assigning a syntactic category to the elements of W. The members of K are atomic symbols.

Example: The sentence *Kogoś tam widział wczoraj?* is triply ambiguous. If *Kogoś tam* is interpreted together as the accusative form of the compound pronoun 'ktoś tam' ('somebody or other'), *widział* – as the 3rd person singular masculine past form of the verb 'widzieć' ('to see'), *wczoraj* as the adverb 'yesterday', the sentence means 'Did he see somebody or other yesterday?'. If *Kogoś* is interpreted as the accusative form of the pronoun 'ktoś' ('somebody'), *tam* – as the adverb 'there', *widział* and *wczoraj* – as above, the meaning is 'Did he see anybody there yesterday?'. If *Kogoś* is interpreted as the accusative form of the pronoun 'kto' ('who') suffixed with the morpheme 'ś' (belonging to the verb, indicating 2nd person singular), *tam* – as above, *widział* – as the form of the verb 'widzieć' without the person/number morpheme, and *wczoraj* – as above, the meaning is in turn 'Who did you see there yesterday?'. The word *tam*, disregarding the context, may be also interpreted as the plural genitive form of the noun 'tama' ('a dike').

The three possible readings correspond to the paths, respectively: (w_1,w_7,w_9), (w_2,w_5,w_7,w_9), and (w_3,w_4,w_5,w_8,w_9) in the morphological graph in Fig. 1.

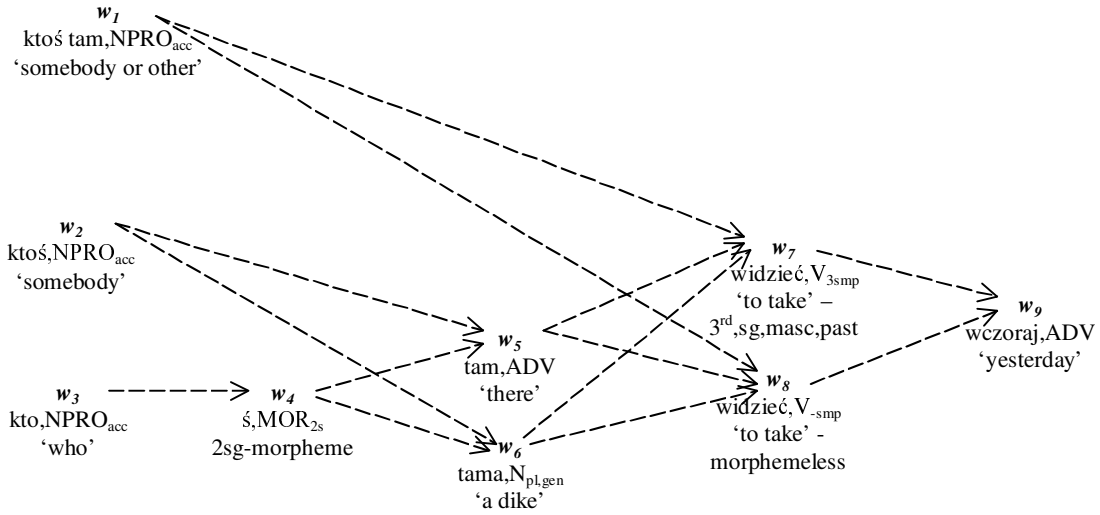

Fig. 1 Morphological graph for the sentence *'Kogoś tam widział wczoraj?'*

3 Dependency graph

The dependency graph represents ambiguous syntactic description with the ambiguities of the other types still unresolved. It is constructed for a given morphological graph, and a given dependency-type grammar in the format described below.

The dependency graph has the same set of nodes as the morphological graph. Its arcs represent possible syntactic dependencies between possible word interpretations contained in *W*. Formally, a dependency graph is a triple $<W,\mapsto,c>$, in which *W* and *c* come from the morphological graph $<W,\prec,c>$, and $\mapsto \subseteq W \times W$. Each subgraph of this graph which is a projective[2] tree built on the nodes of a full morphological path will be a possible dependency description of the sentence

On the basis of the relations $\prec$, $<$, and $\mapsto$ we define several additional relations between elements of *W*:

transitive dependency ($\overset{*}{\mapsto}$): $w_i \overset{*}{\mapsto} w_j \Leftrightarrow w_i = w_j \vee \exists_{w_k \in W} w_i \overset{*}{\mapsto} w_k \wedge w_k \mapsto w_j$

rightward dependency ($\rightarrow$)[3]: $w_i \rightarrow w_j \Leftrightarrow w_i \mapsto w_j \wedge w_i < w_j$

leftward dependency ($\leftarrow$): $w_i \leftarrow w_j \Leftrightarrow w_j \mapsto w_i \wedge w_i < w_j$

transitive rightward dep. ($\overset{*}{\rightarrow}$): $w_i \overset{*}{\rightarrow} w_j \Leftrightarrow w_i = w_j \vee \exists_{w_k \in W} w_i \overset{*}{\rightarrow} w_k \wedge w_k \rightarrow w_j$

transitive leftward dep. ($\overset{*}{\leftarrow}$): $w_i \overset{*}{\leftarrow} w_j \Leftrightarrow w_i = w_j \vee \exists_{w_k \in W} w_i \overset{*}{\leftarrow} w_k \wedge w_k \leftarrow w_j$

---

[2] If $w_i \mapsto w_j$ then all w_k between w_i and w_j in linear order ($\prec$) must be transitive dependents of w_i *or* w_j.

[3] Note the lack of the vertical line at the beginning of the arrow.

visibility relation (~) : $$w_i \sim w_j \Leftrightarrow \exists_{w_k, w_l \in W} \; w_i \overset{*}{\rightarrow} w_k \wedge w_k \prec w_l \wedge w_l \overset{*}{\leftarrow} w_j$$

The visibility relation plays the key role here. It expresses the local condition for two nodes to be connectable, retaining the projectivity condition. In a projective dependency tree this relation holds between each pair of nodes connected by a dependency. Also, a tree in which the visibility relation holds between each pair of connected nodes is projective.

We use the terms 'head' and 'dependent' for words connected by a dependency. Similarly we say 'transitive head' and 'transitive dependent' in the case of a transitive dependency relation. Below we illustrate schematically the notions introduced:

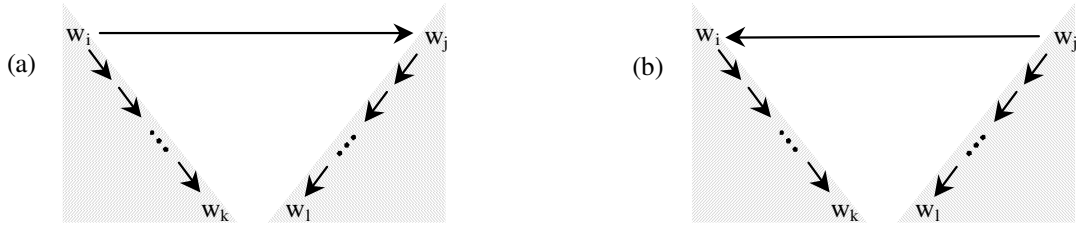

Fig. 2 Subtrees containing all nodes which in linear order are between two connected nodes.

In the subtrees (a) and (b) in Fig. 2 w_i is a transitive left head of w_k, w_k is a transitive right dependent of w_i, w_l is a transitive left dependent of w_j, w_j is a transitive right head of w_l. The visibility relation holds between w_i and w_j. We will say that w_i is visible from w_j on the left, w_j is visible from w_i on the right. (What is more, all the nodes in the path from w_i to w_k are visible on the left from all the nodes in the path from w_j to w_l and vice versa – on the right.)

4 Dependency graph construction (grammar SDG_0)

We will first present the dependency graph construction algorithm for a very simple (and weak) type of dependency grammar, which we will call SDG_0 (SDG=simple dependency grammar). A grammar of type SDG_0 allows only for rules (1) $k_1 \xrightarrow{l} k_2$, (2) $k_1 \xrightarrow{r} k_2$ stating that a word of category k_2 may be a left (1) or right (2) dependent of a word of category k_1. The terms 'left' and 'right' refer to the linear order defined by the relation $\prec$. Formally, having done a nonempty set K of syntactic categories (atomic symbols), a grammar $\mathbf{G}$ of type SDG_0 is defined as a pair $<\mathbf{R},\mathbf{T}>$, where $\mathbf{R} \subseteq K \times K \times \{\mathbf{l},\mathbf{r}\}$ is a set of syntactic rules and $\mathbf{T} \subseteq K$ – is a set of possible root categories.

The algorithm constructs the relation $\mapsto$ (the set of arcs of the dependency graph) for a given morphological graph $<W,\prec,c>$ and a grammar $\mathbf{G} =< \mathbf{R},\mathbf{T} >$. The algorithm does not follow alternative paths, it deterministically builds an ambiguous structure.

First, we define a linear order on elements of W, and assign them indexes from 1 to n=|W| in such a way that $w_i < w_j \Rightarrow i < j$.

The set of arcs $\mapsto$ is constructed incrementally. In *i*-th step we add all possible arcs <w_k,w_i> and <w_i,w_k> for all $k < i$; all arcs between the elements of W with indexes lower than *i* are already known and will not change. An arc $< w_k, w_i >$ ($< w_i, w_k >$) is added to $\mapsto$ if (1) w_k is visible on the left from w_i and (2) there exists a grammar rule $c(w_k) \xrightarrow{r} c(w_i)$ ($c(w_i) \xrightarrow{l} c(w_k)$).

We define the following sets, which will be used in the algorithm:

The set of indexes of transitive left heads of w_i: $H_i = \left\{ k \in \{1,\ldots,n\} \mid w_k \overset{*}{\rightarrow} w_i \right\}$

The set of indexes of right heads of of w_i: $h_i = \left\{ k \in \{1,\ldots,n\} \mid w_i \leftarrow w_k \right\}$

The set of indexes of transitive left dependents of w_i: $D_i = \left\{ k \in \{1,\ldots,n\} \mid w_k \overset{*}{\leftarrow} w_i \right\}$

The set of indexes of right dependents of w_i: $d_i = \left\{ k \in \{1,\ldots,n\} \mid w_i \rightarrow w_k \right\}$

The set of indexes of nodes visible from of w_i on the left: $V_i = \{ k \in \{1,\ldots,n\} \mid w_k \sim w_i \}$

The set of indexes of left neighbours of w_i: $L_i = \{ k \in \{1,\ldots,n\} \mid w_k \prec w_i \}$

Only the sets L_i, H_i and V_i are needed, however, to construct the graph. The sets L_i are known a priori: they follow from the relation $\prec$. The sets H_i and V_i are computed in *i*-th step of the algorithm (see next page). During the graph construction we compute additionally the sets D_i, h_i, and d_i which will be used in the tree construction algorithm (see section 5).

The theoretical worst-case time complexity in the function of the input length is O(n^3) (assuming linear complexity of the set union operation). It directly depends on the number of all possible morphological interpretations of input words (n=|*W*|). This number is proportional to the length of the input. The implementational complexity depends on the data structure used to represent the sets. We use bit vectors for this purpose, so we can assume the unary complexity of the set union operation. In consequence, the practical time complexity of the whole algorithm reduces to O(n^2).

Algorithm 1. Dependency graph construction for grammars of type SDG_0.

1. $\mapsto = \varnothing$
2. for i from 1 to n do
3. $H_i := \{i\}$
4. $V_i := \bigcup_{k \in L_i} H_k$
5. ($D_i := \{i\}$)
6. for j from i-1 down to 1 do
7. if $j \in V_i$
8. if $c(w_i) \xrightarrow{l} c(w_j) \in \mathbf{R}$ // checking w_j as dep.
9. $\mapsto := \mapsto \cup \{< w_i, w_j >\}$
11. $V_i := V_i \cup V_j$
10. ($D_i := D_i \cup D_j$; $h_j := h_j \cup \{i\}$)
12. if $c(w_j) \xrightarrow{r} c(w_i) \in \mathbf{R}$ // checking w_j as head
13. $\mapsto := \mapsto \cup \{< w_j, w_i >\}$
14. $H_i := H_i \cup H_j$
15. ($d_j := d_j \cup \{i\}$)

Comments to the algorithm:

1. The set of arcs is empty.
2. We consider the elements of W in the order of growing indexes.
3. In the set of transitive left heads H_i we have i as the only element in the beginning.
4. The nodes visible on the left from w_i are, initially, the transitive left heads of the left neighbours of w_i (w_i does not yet have dependents).
5. In the set of transitive left dependents D_i we have i as the only element in the beginning.
6. We consider the nodes w_j with indexes from i-1 down to 1 ...
7. ... which are visible from w_i on the left.
8. We check whether the grammar allows to connect w_j to w_i as a dependent.
9. If so, we add the arc from w_i to w_j to the graph.
10. With this arc added, the nodes visible on the left from w_j (which is a left dependent of w_i) are also visible on the left from w_i. We add the elements of V_j to V_i.
11. Similarly, the transitive left dependents of w_j are also transitive left dependents of w_i. We add the elements of D_j to D_i. We add i to the set of right heads of w_j.
12. We check whether the grammar allows to connect w_j to w_i as the head.
13. If so, we add the arc from w_j to w_i to the graph.
14. With this arc added, the transitive left heads of w_j (which is a left head of w_i) are also transitive left heads of w_i. We add the elements of H_j to H_i.
15. We add i to the set of right dependents of w_j.

The important properties of a dependency graph are:

Property 1 For all arcs $<w_i,w_j>$ there exists a subgraph which, together with this arc, constitutes a partial projective dependency tree built on nodes of a partial morphological path between w_i and w_j (inclusive).

This follows from the fact that arcs are always added between pairs of nodes visible one from another (line 7 of the algorithm). So for each arc, we are sure to be able to construct the part of a projective tree which is 'below this arc'. Also, as we have all arcs which are locally legitimate (i.e. taking into account only nodes between the arc's end nodes), we are sure to have all arcs which are globally legitimate (i.e. taking into account the dependency description of the whole sentence). It follows that:

Property 2 All projective dependency trees which can be built on the nodes of a full morphological path are subgraphs of the dependency graph.

Fig 3. shows the dependency graph for the sentence '*Kogoś tam widział wczoraj?*'.

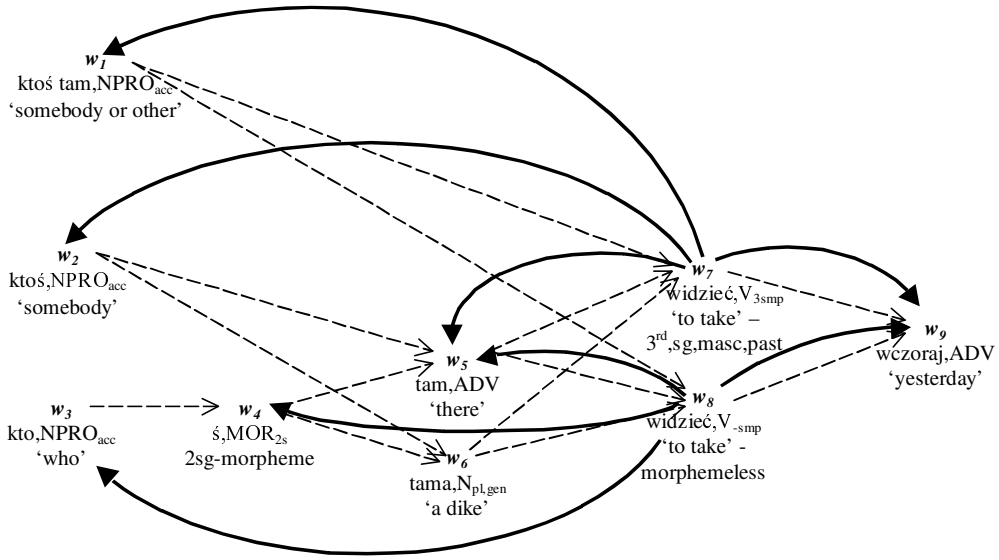

Fig. 3 The dependency graph (solid lines) for the sentence '*Kogoś tam widział wczoraj ?*' with the repeated morphological graph (dashed lines).

5 Finding possible root nodes

A node w_r is the root of a projective dependency tree describing a whole sentence if it is a transitive left head of a final node w_z, an initial node w_a is its transitive left dependent, and $c(w_r) \in \mathbf{T}$.

Let w_a be an initial node such that $w_a \overset{*}{\leftarrow} w_r$. Consider the sequence of arcs $w_a \leftarrow w_b \leftarrow \ldots \leftarrow w_q \leftarrow w_r$. We know (Property 1.), that for each arc $w_i \leftarrow w_j$ in this sequence there exists a projective dependency tree built on the elements of a

morphological path from w_i to w_j. Together they constitute a projective dependency tree on the elements of a morphological path from w_a to w_r. Similarly, we can show that there exists a projective dependency tree on the elements of a morphological path from w_r to w_z. So there exists a tree on the elements of a path from w_a to w_z with the root w_r.

Let us denote by *FIRST* the set of indexes of the initial nodes from *W* (without a predecessor in the morphological graph), by *LAST* – the set of indexes of the final nodes, and by *ROOT* – the set of indexes of the root nodes. To find the elements of *ROOT* we first find transitive left heads of final nodes (the set S_1 in the algorithm below), then we choose those which have as a transitive left dependent an initial node (the set S_2). Finally, we discard the roots not allowed by the grammar.

Algorithm 2. Construction of the set of root nodes.

1. $S_1 := \varnothing$; $S_2 := \varnothing$; $ROOT := \varnothing$
2. for all $i \in LAST$ do $S_1 := S_1 \cup H_i$
3. for all $i \in S_1$ do if $D_i \cap FIRST \neq \varnothing$ then $S_2 := S_2 \cup \{i\}$
4. for all $i \in S_2$ do if $c(w_i) \in \mathbf{T}$ then $ROOT := ROOT \cup \{i\}$

If $ROOT = \varnothing$, no complete dependency tree can be constructed. The time complexity of the algorithm is O(n^2) (or O(n) assuming the unary complexity of the set operations).

6 Construction of dependency trees

Three types of subgraphs are used in the dependency tree construction algorithm:

$\mathbf{A}(i, j)$ – a subtree between w_i and w_j such that $w_i \overset{*}{\leftarrow} w_j$.

$\mathbf{B}(i, j)$ – a subtree between w_i and w_j such that $w_i \overset{*}{\rightarrow} w_j$.

$\mathbf{C}(i, j)$ – a subgraph between w_i and w_j visible one from another.

We use the expression ‘a subtree (subgraph) between w_i and w_j’ to refer to a subtree (subgraph) containing w_i, w_j and all nodes between them in linear order.

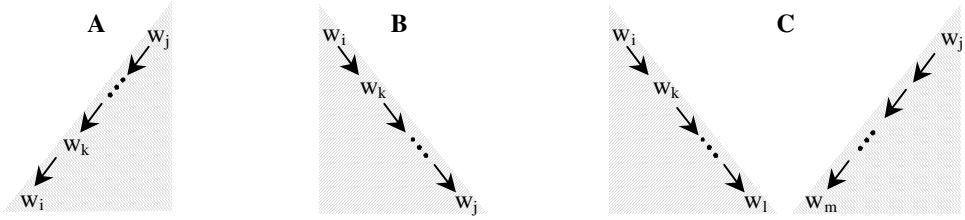

Fig. 4 Subgraphs of types A, B, and C

The tree construction is sketched in the form of a recursive ideterminismtic algorithm. Alternative trees can be generated through backtracking. It should be noted that backtracking is not used to find a solution, but to generate the next solution. Each choice is successful. Also, each possible tree will be generated exactly once.

Algorithm 3. Dependency tree construction

1. $\mapsto := \varnothing$
2. choose an $r \in ROOT$ // choose the root node
3. choose an $a \in FIRST \cap D_r$ // choose the initial node
4. choose a $z \in LAST$ such that $r \in H_z$ // choose the final node
3. construct $\mathbf{A}(a, r)$ and $\mathbf{B}(r, z)$

$\mathbf{A}(i, j)$ construction

if *i*=*j*, done, otherwise choose a $k \in h_i \cap D_j$ // $w_i \leftarrow w_k \ \wedge \ w_k \overset{*}{\leftarrow} w_j$

$\mapsto = \mapsto \cup \{< w_k, w_i >\}$

construct $\mathbf{C}(i,k)$ and $\mathbf{A}(k, j)$

$\mathbf{B}(i, j)$ construction

if *i*=*j*, done, otherwise choose a $k \in d_i \cap H_j$ // $w_i \rightarrow w_k \ \wedge \ w_k \overset{*}{\rightarrow} w_j$

$\mapsto = \mapsto \cup \{< w_i, w_k >\}$

construct $\mathbf{C}(i,k)$ and $\mathbf{B}(k, j)$

$\mathbf{C}(i, j)$ construction (R_i is a set of right neighbours of w_i.)

if $i \in L_j$, done, otherwise // $w_i \prec w_j$

either choose a $k \in d_i \cap V_j$ // $w_i \rightarrow w_k \ \wedge \ w_k \sim w_j$

$\mapsto = \mapsto \cup \{< w_i, w_k >\}$,

construct $\mathbf{C}(i,k)$ and $\mathbf{C}(k, j)$

or choose a $k \in R_i \cap D_j$ // $w_i \prec w_k \ \wedge \ w_k \overset{*}{\leftarrow} w_j$

construct $\mathbf{A}(k, j)$

The time complexity of the algorithm is $O(n^2)$ (or $O(n)$ if we assume the unary complexity of the set intersection operation), where n is the number of arcs in the dependency tree being constructed. Each of the subroutines constructing **A**, **B**, and **C** adds one arc or returns (the subroutine **C** adds an arc or calls the subroutine **A**, which adds an arc), so each may be called at most $2n$ times.

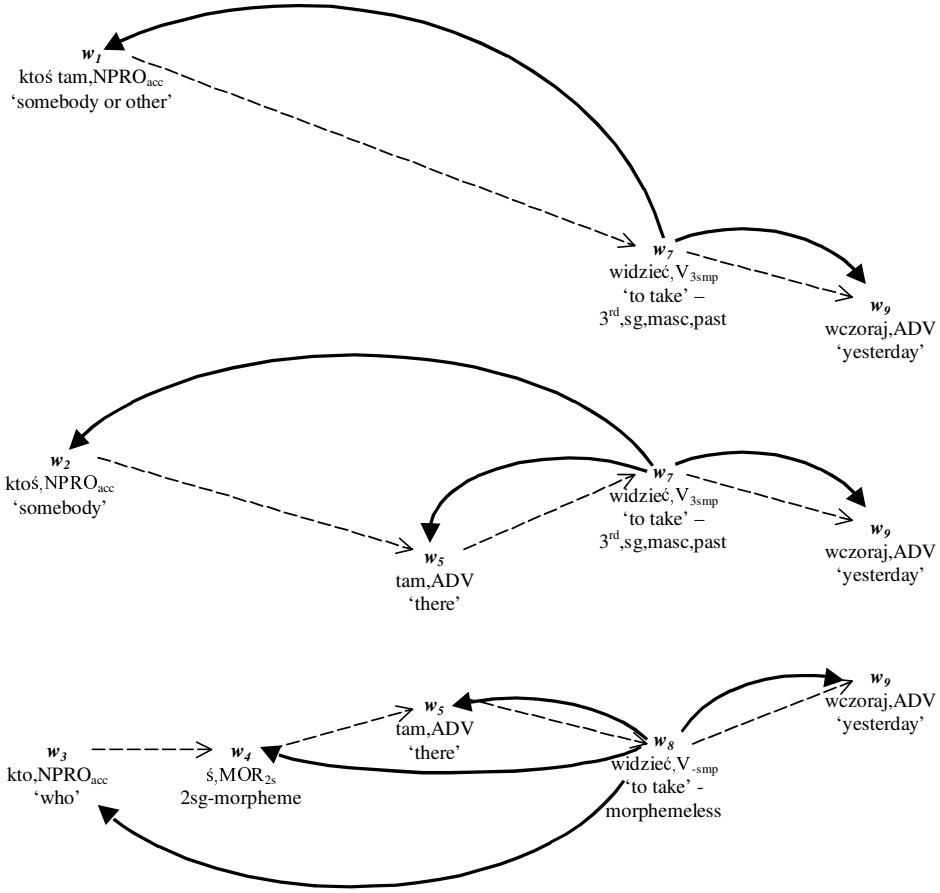

Fig. 5 Three possible dependency trees constructed from the dependency graph in Fig. 3

7 Grammar SDG_1

Grammars of type SDG_0 are too weak to be practically applicable. We introduced them in order to present the algorithms in its basic form. We will extend the grammar format with rules indicating that certain dependents are obligatory. If we limit the number of obligatory dependents of a given head category to at most one at each side, the complexity bounds of the algorithms remain the same.

A grammar of type SDG_1 contains rules of four types :

(1) $k_1 \xrightarrow{l}_{fac} k_2$, (2) $k_1 \xrightarrow{r}_{fac} k_2$ - a word of category k_2 may be a left (right) dependent of a word of category k_1;

(3) $k_1 \xrightarrow{l}_{obl} k_2$, (4) $k_1 \xrightarrow{r}_{obl} k_2$ - a word of category k_2 must be a left (right) dependent of a word of category k_1;

The rules of types (1) and (2) may be applied without any constraints, as before. The application of rules (3) and (4) is constrained in the following way: for each node assigned the category k_1 exactly one rule of type (3) and one of type (4) have to be applied, unless no such rules for this category are defined. A node for which this constraint is satisfied will be called *saturated*.

A grammar G of type SDG_1 is defined as a pair **<R,T>**, where $\mathbf{R} \subseteq K \times K \times \{\mathbf{l},\mathbf{r}\} \times \{\mathbf{fac},\mathbf{obl}\}$ and $\mathbf{T} \subset K$ – is a set of possible root categories.

Example: The language $a^n b^n$ may be described with a grammar of type SDG_1 (a is the syntactic category for the symbol a, and b for b).

$$R = \left\{ a \xrightarrow{r}_{obl} b,\ b \xrightarrow{l}_{fac} a \right\}, \quad T = \{a\}$$

The properties of the dependency graph are now stated as follows:

Property 1' For all arcs $<w_i,w_j>$ ($<w_j,w_i>$), where $w_i<w_j$, there exists a subgraph which, together with this arc, constitutes a partial projective dependency tree built on the nodes of a partial morphological path from w_i to w_j in which all nodes between w_i and w_j are saturated, and w_j is left-saturated (w_i is right-saturated).

Property 2' All projective dependency trees in which all nodes are saturated and which can be built on the nodes of a full morphological path, are subgraphs of the dependency graph.

In order to take into account the saturation information H_i is replaced by two sets: H_i^+ – the set of right-saturated transitive left heads and H_i^- – the set of right-unsaturated transitive left heads. Similarly, D_i is replaced by D_i^+ – the set of transitive left dependents of left-saturated w_i and D_i^- – the set of transitive left dependents of left-unsaturated w_i. V_i is replaced by tree sets: V_i^{++} – the set of right-saturated nodes visible from left-saturated w_i, V_i^{+-} – the set of right-saturated nodes visible from left-unsaturated w_i, V_i^{-+} – the set of right-unsaturated nodes visible from left-saturated w_i. The sets h_i and d_i are also duplicated to store separately information about facultative and obligatory connections. In Fig 6. an arrow drawn with a dashed line going from a node to outside the shaded triangle (representing a subtree) indicates that the node is unsaturated at the respective side. All other nodes of the subgraphs shown in the figure are saturated.

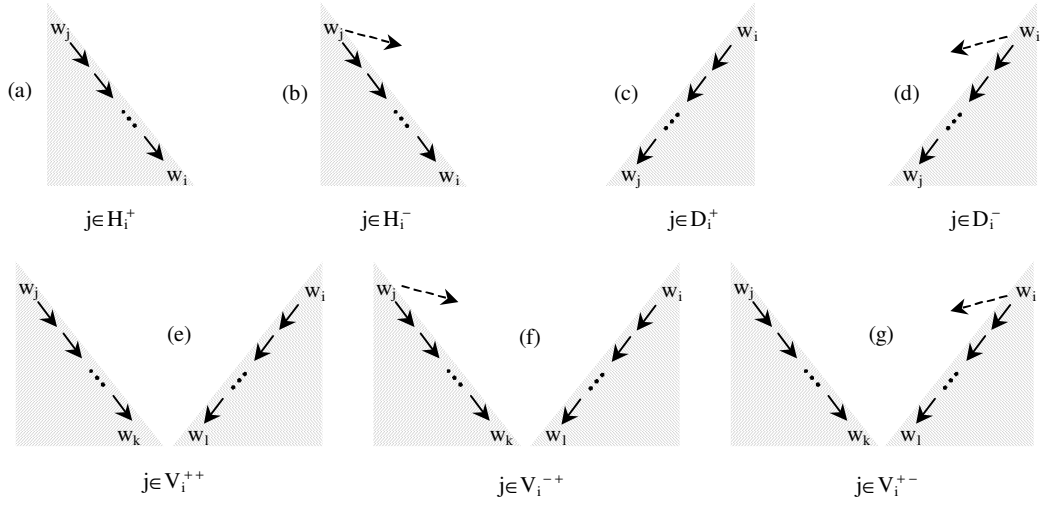

Fig. 6. Set membership illustration

Below we show how the sets H_i^+, H_i^-, V_i^{++}, V_i^{+-}, and V_i^{-+} are computed in the internal loop.

Algorithm 4. Dependency graph construction algorithm (SDG_1) – the internal loop

1. for j from i-1 down to 1 do
2. if $j \in V_i^{++} \cup V_i^{+-}$ // checking w_j as dep.
3. if $c(w_i) \xrightarrow{l}_{obl} c(w_j) \in \mathbf{R}$ and $j \in V_i^{+-}$ // obligatory
4. $\mapsto := \mapsto \cup \{< w_i, w_j, \mathbf{obl} >\}$
5. $V_i^{++} := V_i^{++} \cup V_j^{++}$
6. $V_i^{-+} := V_i^{-+} \cup V_j^{-+}$
7. if $c(w_i) \xrightarrow{l}_{fac} c(w_j) \in \mathbf{R}$ // facultative
8. $\mapsto := \mapsto \cup < w_j, w_i, \mathbf{fac} >$
9. if $j \in V_i^{++}$ // w_i left-saturated
10. $V_i^{++} := V_i^{++} \cup V_j^{++}$
11. $V_i^{-+} := V_i^{-+} \cup V_j^{-+}$
12. if $j \in V_i^{+-}$ // w_i left-unsaturated
13. $V_i^{+-} := V_i^{+-} \cup V_j^{++}$
14. if $j \in V_i^{++} \cup V_i^{-+}$ // checking w_j as head
15. if $c(w_j) \xrightarrow{r}_{obl} c(w_i) \in \mathbf{R}$ and $j \in V_i^{-+}$ // w_i as oblig. dep.
16. $\mapsto := \mapsto \cup < w_j, w_i, \mathbf{obl} >$
17. $H_i^+ := H_i^+ \cup H_j^+$
18. $H_i^- := H_i^- \cup H_j^-$
19. if $c(w_j) \xrightarrow{r}_{fac} c(w_i) \in \mathbf{R}$ // w_i as facult. dep.
20. $\mapsto := \mapsto \cup < w_j, w_i, \mathbf{fac} >$
21. if $j \in V_i^{++}$ // w_j right-saturated
22. $H_i^+ := H_i^+ \cup H_j^+$
23. $H_i^- := H_i^- \cup H_j^-$
24. if $j \in V_i^{-+}$ // w_j right-unsaturated
25. $H_i^- := H_i^- \cup \{j\}$

The other algorithms can also be modified to deal with the saturation information.

8 Conclusion

We have presented an approach to dependency parsing with good computational properties. We have shown how instead of a pars forest a single graph with a constant set of nodes may be used to store alternative dependency structures. The price we pay for the use of the effective structure sharing method is the restricted descriptive power of grammars which can be handled.

Our approach may be interesting for the application fields where efficiency considerations are important and simplified syntactic analysis is acceptable. In particular, we mean here the area of large corpora processing, including location of phrases of a specific type, syntactic concordancing, syntactic and lexical information retrieval, and morphological disambiguation.

Significant work has still to be done to bring these ideas to practice. The main directions we see for future work are: grammar format elaboration (e.g. to allow a dependent required on either side), incorporation of punctuation information, elaboration of algorithms for more thorough dependency graph analysis (e.g. identification of useless nodes and arcs). Also, it would be interesting to examine how the probabilistic information could be used to indicate preferred parses.

References

[1] J. Courtin, D. Genthial, Parsing with Dependency Relations and Robust Parsing, in: Kahane S. and Polguere A. (eds.), *Processing of Dependency-based Grammars: Proceedings of the Workshop*, COLING-ACL'98, Montreal (1998)

[2] P. Hellwig, Chart Parsing according to the slot and filler principle, *Proceedings of COLING-88*, Budapest (1988) 242-244

[3] I.A. Mel'čuk , *Dependency Syntax: Theory and Practice*, State University of New York Press, New York (1988)

[4] M. Silberztein, *Dictionnaires electroniques et analyse automatique de textes. Le systeme INTEX*, MASSON, Paris (1993)

[5] Tesnière L., *Eléments de syntaxe structurale*, Klincksiek, Paris (1959)