2005－ FI － 80（15）
2005－ NL － 169（15）
2005／9／30

Graph Branch Algotithm: An Optimum Tree Search Method for Scored Dependency Graph with Arc Co-occurrence Constraints

Hideki Hirakawa
Knowledge Media Laboratory
TOSHIBA Corporate Research & Development Center
1, Komukai-Toshiba-cho, Saiwaiku, Kawasaki, 212-8582, Japan
email: hideki.hirakawa@toshiba.co.jp

Preference Dependency Grammar (PDG) is a framework for the morphological, syntactic and semantic analysis of natural language sentences. PDG gives packed shared data structures for encompassing the various ambiguities in each levels of sentence analysis with preference scores and a method for calculating the most plausible interpretation of a sentence. This paper proposes the Graph Branch Algorithm for computing the optimum dependency tree (the most plausible interpretation of a sentence) from a scored dependency forest which is a packed shared data structure encompassing all possible dependency trees (interpretations) of a sentence. The graph branch algorithm adopts the branch and bound principle for managing arbitral arc co-occurrence constraints including the single valence occupation constraint which is a basic semantic constraint in PDG.

1 Introduction

Dependency representation as well as phrase structure representation is a basic framework used for various kinds of NLP applications As described in Ref.1), various dependency parsing methods are proposed. Some methods utilize lexicalized phrase-structure parsers with the ability to output dependency information[2),3)] and some methods obtain dependency trees directly[4)~8)]. Many of dependency parsers generate only projective dependency trees but some parsers treat non-projectivity[1)].

Training corpuses and statistical information are used for computing the most appropriate dependency tree in many parsers. One class of parsers choose the optimum decision during parsing process[4),8)]. Another class of parsers generate a dependency graph encompassing all possible dependency trees for a sentence and searches for the optimum tree[1),5),6),10)]. Generally, the total score of a dependency tree is defined as sum total of scores of dependency arcs in it.

Ref.5), 6) adopts dynamic programming principle for searching the optimum tree from a dependency graph containing WPP [*3] nodes. Ref.1) treats a dependency graph containing word nodes and search the maximum spanning tree with highest score based on the Chu-Liu-Edmonds algorithm or the Eisner's algorithm[7)]. Ref.9) proposed a dependency graph called a “Semantic Dependency Graph” (SDG), which represents ambiguities in word dependencies and their semantic relations. Ref.10) proposed an algorithm for searching the optimum tree from a semantic dependency graph with preference scores based on the branch and bound method[12)]. I call this kind of optimum tree search method the “Graph Branch Method”.

[*3] WPP is a pair of a word and a part of speech (POS). The word “time” has WPPs such as “time/n” and “time/v”.

The sentence analysis method based on the semantic dependency graph is effective because it employs linguistic constraints as well as linguistic preferences. However, this method is lacking in terms of generality in that it cannot handle backward dependency and multiple WPP because it depends on some linguistic features peculiar to Japanese. “Preference Dependency Grammar” is a general sentence analysis framework employing a new data structure called the “Dependency Forest”(DF)[11)] rather than the semantic dependency graph. The dependency forest is a packed shared data structure which encompasses all dependency trees corresponding to parse trees in a packed shared parse forest[13)] for a sentence. The dependency forest has none of the language-dependent premises that the semantic dependency graph has, so it is applicable to English and other languages. PDG has one more advantage that it can generate non-projective dependency trees because the mapping from phrase structure to dependency structure is defined in grammar rules.

The optimum tree search algorithm for a semantic dependency graph is not applicable to the dependency graph. This paper gives a brief explanation of the dependency forest and shows the graph branch algorithm for obtaining the optimum solution (tree) in the dependency forest.

2 SDG and DF

2.1 SDG and its Drawbacks

Fig.1 shows a semantic dependency graph for “Watashi-mo Kare-ga Tukue-wo Katta Mise-ni Utta”[10)]. The nodes in the graph correspond to the content words in the sentence and the arcs show possible semantic dependency relations between the nodes. Each arc has an arc ID and a preference score. Interpretations of a sentence are well-formed spanning trees that satisfy two constraints, i.e., no cross dependency and no multiple valence occupation. The score of an interpretation is the sum total of arc scores in a semantic dependency tree. The bold arcs in the graph in Fig.1 shows the optimum interpretation with a maximum score of 130.

The semantic dependency graph is designed based on the Japanese kakari-uke relation and assumes the following features of Japanese.

(a) A dependant always locates to the left of its governor (no backward dependency)
(b) POS ambiguities are quite minor compared with English *4

The semantic dependency graph and its optimum solution search algorithm adopt these as their premises. Therefore, this method is inherently inapplicable to languages like English that require backward dependency and multiple POS analysis.

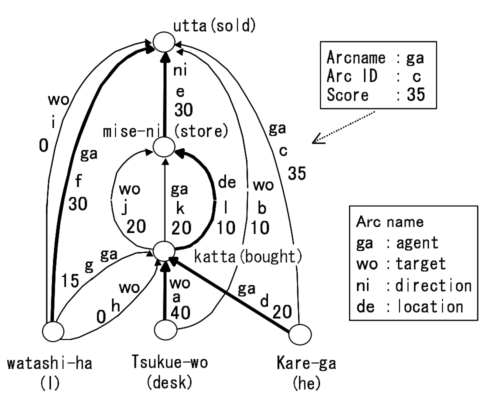
2.2 DF and Optimum Tree

2.2.1 Overview of DF

The dependency forest is a packed shared data structure encompassing all possible dependency trees for a sentence. The dependency forest consists of a dependency graph (DG) and a co-occurrence matrix (CM). Fig.2 shows a dependency graph for the example sentence “Time flies like an arrow.”

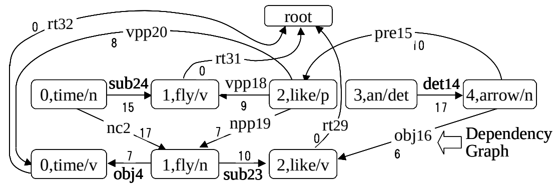
The dependency graph consists of nodes and directed arcs. A node represents a WPP and an arc shows the dependency relation between nodes. An arc has its ID and preference score. CM is a matrix whose rows and columns are a set of arcs in DG that prescribes the co-occurrence relation between arcs. Only when CM(i,j) is ○, arc_i and arc_j are co-occurrable in one dependency tree.

The dependency forest has correspondence with the packed shared parse forest*5. This means that the dependency forest provides a means to treat all possible interpretations of a sentence in dependency structure representation.



Optimum Semantic Dependency Tree: [a, d, e, f, l]

Fig.1: Example of semantic kakari-uke graph and its optimum solution



Meaning of Arc Name
sub : subject
obj : object
npp : noun-preposition
vpp : verb-preposition
pre : preposition
nc : noun compound
det : determiner

Co-occurence Matrix

	2	24	4	23	19	18	20	14	16	15	31	29	32
2	–			○				○	○			○	
24		–				○		○		○	○		
4			–		○		○	○		○			○
23	○			–				○	○			○	
19			○		–			○		○			○
18		○				–		○		○	○		
20			○				–	○		○			○
14	○	○	○	○	○	○	○	–	○	○	○	○	○
16	○			○				○	–			○	
15		○	○		○	○	○	○		–	○		○
31		○				○		○		○	–		
29	○			○				○	○			–	
32			○		○		○	○		○			–

Fig.2: Score-added DF for “Time flies like an arrow”

2.2.2 Well-formed Dependency Tree

One sentence interpretation is represented by one well-formed dependency tree which satisfies the following well-formed dependency tree conditions[11)]:

(a) No two nodes occupy the same input position (single role constraint)
(b) Every input word has the corresponding node in the tree (coverage constraint)
(c) Each arc pair in a tree has a co-occurrence relation in CM (co-occurrence constraint)

Conditions (a) and (b) are collectively referred to as “covering condition”. A dependency tree that satisfies the covering condition is called a well-covered dependency tree. In semantic dependency graphs, a spanning tree of a graph is a well-covered tree. This simplifies the development of an optimum solution search algorithm. The algorithm for the dependency forest requires the concept of covering condition.

2.3 Relation Between SDG and DF

Nodes in a dependency forest are a set of nodes in the WPP trellis produced from an input sentence whereas nodes in a semantic dependency graph are a set of nodes forming a path in the WPP trellis, i.e., a subset of the dependency graph. Therefore, the dependency forest contains the semantic dependency graph. On the other hand, well-formedness constraints introduced to a semantic graph, i.e. the cross dependency and multiple valence occupation constraints, are a type of arc co-occurrence constraints representable by means of a co-occurrence matrix. Therefore, the dependency forest is a generalized and more powerful data structure covering the representative power of the semantic dependency graph.

3 Optimum Tree Search

The graph branch method works on the branch and bound principle and searches the optimum well-formed tree from a dependency graph by applying partial sub-problem expansions called graph branching. The algorithm in Ref.10) applies the graph branch method to the semantic dependency

*4 Word boundary ambiguity corresponding to the compound word boundary ambiguity in English exists in Japanese. Treatment of this ambiguity is a practical problem for the semantic dependency graph even when applied to Japanese sentence analysis

*5 The correspondence between the parse tree and the dependency tree is generally 1 to N and vice versa.

graph. Unfortunately, this algorithm is not directly applicable to the dependency forest search problem. The following shows a new algorithm for applying the graph branch method to the dependency forest.

3.1 Branch and Bound Method

The branch and bound method is a principle for solving computationally hard problems such as NP-complete problems. The basic strategy is that the original problem is decomposed into easier partial-problems (branching) and the original problem is solved by solving them. Pruning called a bound operation is applied if it turns out that the optimum solution to a partial-problem is inferior to the solution obtained from some other partial-problem (dominance test), or if it turns out that a partial-problem gives no optimum solutions to the original problem (maximum value test). The dominance test is not used in the graph branch method. Usually, the branch and bound algorithm is constructed to minimize the value of the solution. The graph branch algorithm in this paper is constructed to maximize the score of the solution because the best solution is the maximum tree in the dependency forest.

The following features for the maximum bound value test with respect to the problem P and its partial-problem P_c must be satisfied in the branch and bound method.

(MC1) $g(P_c) \geq f(P)$ where $g(P_c)$ is the maximum value of P_c, and $f(P)$ is the maximum value of P.

(MC2) If $g(P_c) = l(P)$ where l gives a value of a feasible solution to P, then the feasible solution is a solution to P.

(MC3) If P_c has no feasible solutions then P has no solutions.

(MC4) If a feasible solution with an incumbent value z is obtained for some partial-problem, and if $g(P_c) \leq z$, then partial-problems branched from problem P have no better solutions than z.

These conditions are called model conditions in this paper. In the case of MC2-MC4., partial-problem P_c can be terminated. Fig.3 shows a general branch and bound algorithm for obtaining one optimum solution[12)].

3.2 Graph Branch Algorithm

Fig.3 shows a skeleton of the algorithm. In order to make it running code, each operation in the algorithm must be realized for the target problem. The graph branch algorithm applies the branch and bound method to the optimum tree search problem with the binary arc co-occurrence constraint by introducing the graph branch operation for the partial-problem expansion operation. Fig.4 shows the graph branch algorithm which has been extended from the original skeleton to search all optimum trees for a dependency graph. The following sections explains how the components of the branch and bound method in Fig.3 are implemented in the graph branch algorithm.

A : Set of active partial problems (not yet terminated nor expanded)
N : Set of generated partial problems
O : set of optimum solutions
z : incumbent value
l : l(P) gives value of feasible solution of a partial problem P
g : g(P) gives upper-bound value of a partial problem P
s : s(A) selects one partial-problem in A
G : Set of partial problems with no feasible solution or g(P)=f(P)
f : f(P) is the optimum solution of P
D : If P_i D P_j, P_i dominates P_j

S1(initial value setup): A:={P_0}, N:= {P_0}, z=-∞, O:={}
S2(search) : If A={} goto S9 else Pi:=s(A). Goto S3.
S3(incumbent value update) : If l(Pi)>z then z:=l(Pi), O:={x} (x is a feasible solution of Pi satisfying f(x)≧l(x)). Goto S4.
S4(G test) : If Pi∈G goto S8 else goto S6.
S5(upper bound test) : If g(Pi)≦z goto S8 else goto S6.
S6(dominance test) : If there exists Pk (≠Pi)∈N satisfying Pk D Pi goto S8 else goto S7.
S7(branching operation) : Generate child partial problem Pi_1, Pi_2,..Pi_k of Pi. Set A:=A∪{Pi_1,Pi_2,..Pi_k}−{Pi}, N:=N ∪ {Pi_1,Pi_2,..,Pi_k}. Goto S2.
S8(termination of Pi) : Set A:=A−{Pi}. Goto S2.
S9(stop) : Computation stop. If z=-∞ then P0 has no feasible solutions else z is the optimum value f(P0) and x in O is the optimum solution to P0.

Fig.3: Skeleton of branch and bound algorithm

```
P0 : Initial problem, Pi : Partial problem,
AP: Active partial problem list,
O : Set of incumbent solutions, z : Incumbent value

start: /* S1(initial value setup) */
  AP := {P0}; z = -1; O := {};
  UB = get_ub(P0); /* Upper bound of P0 */

search_top: /* S2(search) */
  if(AP == {}) { goto exit; }
  else{ Pi := select_problem(AP); }
  /* Compute the feasible solution FB and the lower */
  /* bound LB (= the score of FS) for Pi.          */
  (FS,LB) := get_fs(Pi);
  /* If no feasible solution found, terminate the problem. */
  if(FS == no_solution) { goto terminate_problem; }
  /* S3(incumbent value update): If LB is better than z,  */
  /* update incumbent solution and incumbent value.       */
  if(LB > z) { z := LB; O := {FS}; }
  /* S5(upper bound test): */
  if(UB < z) { goto terminate_problem; }
  /* Compute inconsistent arc pair list IAPL. */
  IAPL := get_iapl(Pi);
  /* If lower bound (score of feasible solution) is less */
  /* than upper bound, execute graph branch operation.   */
  if(LB < UB) { BACL := IAPL; goto branch; }
  /* Lower bound equals to upper bound => optimum solution */
  elsif(LB == UB) {
    O := {FS} ∪ O; /* Add this FS as incumbent solution */
    /* S8(search more optimum solutions) */
    /* (a) existence of an inconsistent arc pair */
    if(IAPL != {}) { BACL := IAPL; goto branch; }
    /* (b) existence of a rival arc */
    BACL := arcs_with_alternatives(FS);
    if(BACL != {}) { goto branch; }
    else { goto terminate_problem; } }

branch: /* S6(branching operation) */
  /* Generate child partial problems based on BACL */
  ChildProblemList := graph_branch(Pi,BACL);
  AP := AP U ChildProblemList - {Pi}; goto search_top;

terminate_problem:  /* S7(termination of Pi) */
  AP := AP - {Pi}; goto search_top;
```

Fig.4: Graph Branch Algorithm

3.2.1 Partial-problem

Partial-problem P_i in the graph branch method is a problem searching all the well-formed optimum trees in a dependency forest DF_i consisting of the dependency graph DG_i and co-occurrence matrix CM_i. Partial-problem P_i consists of the following elements.

(a) Dependency graph DG_i
(b) Co-occurrence matrix CM_i
(c) Feasible solution value LB_i (corresponding to $l(P)$ in Fig.3)
(d) Upper bound value UB_i (corresponding to $g(P)$ in Fig.3)
(e) Inconsistent arc pair list $IAPL_i$.

The co-occurrence matrix is common to all partial-problems, so one CM is shared by all partial-problems. DG_i is represented not by arcs in DG_i but by arcs not in DG_i but in the whole dependency graph DG. "$rem[..]$" shows arcs removed from DG. For example, "$rem[b, d]$" represents a partial dependency graph $[a, c, e]$ in the case $DG = [a, b, c, d, e]$. This reduces the memory space and the computation for a feasible solution as described below. $IAPL_i$ is a list of inconsistent arc pairs. An inconsistent arc pair is an arc pair which does not satisfy some co-occurrence constraint.

3.2.2 Algorithm for Feasible Solution and Lower Bound Value

In graph branch method, a well-formed dependency tree in the dependency graph DG of the partial-problem P is assigned as the feasible solution FS (corresponding to x in Fig.3) of P *6. The score of the feasible solution FS is assigned as the lower bound value LB (corresponding to $l(P)$ in Fig.3). The function for computing these values get_fs is called a feasible solution/lower bound value function. Fig.5 shows the algorithm of get_fs. Basically, get_fs searches one feasible solution in higher-score-first and depth-first manner. When an arc which violate co-occurrence constraint against one of the selected arcs is found, get_fs backtracks at *step*5 to the nearest choice point which resolves the contradiction. This assures that the obtained solution satisfies the co-occurrence condition. Furthermore, if get_fs finds no solution, then the problem P has no solution. Since get_fs selects one arc for each position in a sentence, the obtained arcs satisfies the well-covered condition.

Arc groups S_1 to S_n are sorted according to their scores in *step*1. This operation is introduced to obtain a better (higher score) feasible solution, since the better feasible solution lead to a higher incumbent value which bounds more partial-problems.

3.2.3 Algorithm for Obtaining Upper Bound Value

Given a set of arcs A which is a subset of a dependency graph DG, if the set of dependent nodes *7 of arcs in A satisfies the covering condition described above, the arc set A is called the well-covered arc set. The maximum well-covered arc set is defined as a well-covered arc set with the highest score. In general, the maximum well-covered arc set does not satisfy the single role constraint and does not form a tree. In the graph branch method, the score of the maximum well-covered arc set of a dependency graph G is assigned as the upper bound value UB (corresponding to $g(P)$ in Fig.3) of the partial-problem P. Upper bound function get_ub calculates UB by scanning the arc lists sorted by the surface position of the dependent nodes of the arcs.

The above settings satisfy the model conditions. In these settings, P and get_ub corresponds to P_c and $g(P_c)$ respectively. (MC1) is satisfied because $get\_ub(P) \geq f(P)$ is true for $f(P)$ (the score of the optimum tree). (MC2) and (MC4) are satisfied because get_ub is the score of the maximum well-covered arc set. (MC3) is satisfied since $get\_ub(P)$ always has its solution. Therefore, partial-problem P is prunable if the incumbent value z satisfies $z \geq g(P)$ *8.

G: Dependency graph of a partial problem,
n: Number of words in a input sentence
FS: Area for saving arc IDs of a feasible solution
BP: Area for saving the nearest backtrack points
N(S): Number of elements in arc set S
score(FS): Sum total of scores of arcs in FS

step1(grouping and sorting arcs): Classify the arcs in graph G by their starting nodes, and generate the sets of arcs $S_1, S_2, \ldots, S_n$. Sort elements in each S_i with respect to their weights in descending order. Then, sort $S_1, S_2, \ldots, S_n$ with the maximum score of the arcs in the set in descending order. This is renamed $S_1, S_2, \ldots, S_n$。

step2(initialize): FS:=[],BP:=[],I:=1,j:=1,k:=1,l:=0

step3(termination check1): If i>n then terminate by returning the feasible solution FS and its score score(FS). If i≦n then goto step4.

step4(termination check2): If $N(S_i) \geqq j$ then goto step5 else set FS:=no_solution and terminate. (No feasible solution)

step5(constraint check): If $j > N(S_i)$ (no arcs in S_i satisfies the co-occurrence constraint), goto step6. Perform the co-occurrence constraint check between j-th element a(i,j) of S_i and each element $e_1, e_2, \ldots, e_{i-1}$ in FS in reverse order. If a(i,j) does not satisfy the co-occurrence constraint with element e_k ($1 \leqq k \leqq i-1$), set l:=max(l,k), j:=j+1, goto step5. If all co-occurrence constraint checks are satisfied then goto step7.

step6(backtracking): Remove $e_l, e_{l+1}, \ldots, e_{i-1}$ from S. Set j:= BP[l]+1, i:=l. Goto step4.

step7(next node): Add a(i,j) to the last of FS. Set BP[i]:=j, i:=i+1, j:=1. Goto step3.

Fig.5: Algorithm for obtaining FS and LB

---

*6 A feasible solution may not be optimum but is a possible interpretation of a sentence. Therefore, it can be used as an approximate output when the search process is aborted.

*7 The dependent node of an arc is the node located at the source of the arc.

*8 In the case of obtaining all optimum solutions ,the terminate condition should be changed to $z > g(P)$.

3.2.4 Branch Operation

Fig.6 shows a branch operation in the graph branch method called a graph branch operation. Child partial-problems of P are constructed as follows:

(1) Search an inconsistent arc pair (arc_i, arc_j) in the maximum well-covered arc set for the dependency graph of P.
(2) Create child partial-problems P_i, P_j which have new dependency graphs $DG_i = DG - \{arc_j\}$ and $DG_j = DG - \{arc_i\}$ respectively.

Since a solution to P cannot have both arc_i and arc_j simultaneously due to the co-occurrence constraint, the optimum solution of P is obtained from either/both P_i or/and P_j. The child partial-problem is easier than the parent partial-problem because the size of the dependency graph of the child partial-problem is less than that of its parent.

In Fig.4, *get_iapl* computes the list of inconsistent arc pairs $IAPL$(Inconsistent Arc Pair List) for the maximum well-covered arc set of P_i. Then the graph branch function *graph_branch* selects one inconsistent arc pair (arc_i, arc_j) from $IAPL$ for branch operation. The selection criteria for (arc_i, arc_j) affects the efficiency of the algorithm. *graph_branch* selects the inconsistent arc pair containing the highest score arc in $BACL$(Branch Arc Candidates List). *graph_branch* calculates the upper bound value for a child partial-problem by *get_ub* and sets it to the child partial-problem. Simultaneously, *graph_branch* executes bound operation by immediately pruning the child partial-problem whose upper bound value is less than the incumbent value z.

3.2.5 Selection of Partial-problem

select_problem in Fig.5 corresponds to the search $s(A)$ in Fig.3. The best bound search is employed for *select_problem*, i.e. it selects the partial-problem which has the maximum bound value among the active partial-problems. It is known that the number of partial-problems decomposed during computation is minimized by this strategy in the case that no dominance tests are applied[12].

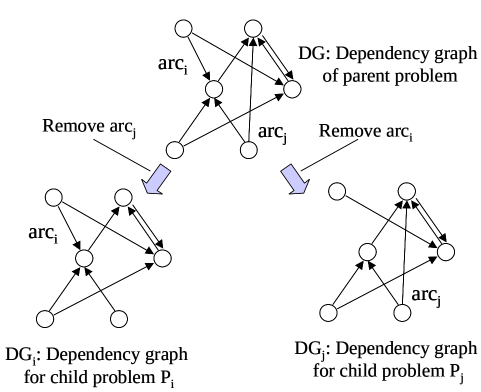


Fig.6: Graph Branching

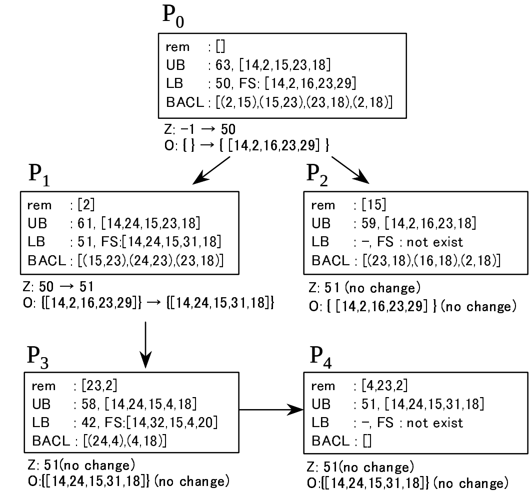


Fig.7: Search diagram for the example sentence

3.2.6 Searching All Optimum Trees

In order to obtain all optimum solutions, partial-problems whose upper bound values are equal to the score of the optimum solution(s) are expanded at $S8(SearchMoreOptimumSolutions)$. In the case that at least one inconsistent arc pair remains in a partial-problem (i.e. $IAPL \neq \{\}$), graph branch is performed based on the inconsistent arc pair. Otherwise, the obtained optimum solution FS is checked if one of the arcs in FS has an equal rival arc. The equal rival arc of arc A is an arc whose position and score are equal to those of arc A. If an equal rival arc of an arc in FS exists, a new partial-problem is generated by removing the arc in FS. $S8$ assures that no partial-problem has an upper bound value greater than or equal to the score of the optimum solutions when the computation stopped.

4 Example of Optimum Tree Search

This section presents an example showing the behavior of the graph branch algorithm using the dependency forest in Fig.2 and some typical ambiguous sentences.

4.1 Example of Graph Branch Algorithm

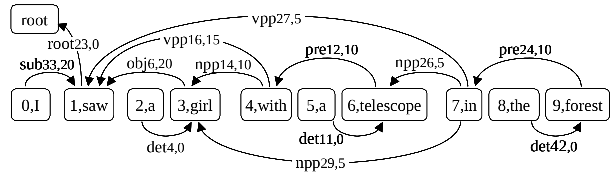
The search process of the branch and bound method can be shown as a search diagram constructing a partial-problem tree representing the parent-child relation between the partial-problems. Figure 7 is a search diagram for the example dependency forest showing the search process of the graph branch method.

In this figure, box P_i is a partial-problem with its dependency graph rem, upper bound value UB, feasible solution and lower bound value LB and inconsistent arc pair list $IACL$. Suffix$i$ of P_i indicates the generation order of partial-problems. Updating of global variable z (incumbent value) and O (set of incumbent solutions) is shown under the box. The value of the left-hand side of the arrow is updated to that of right-hand side of the arrow during the partial-problem processing. Details of the behavior of the algorithm in Fig.4 are described below.

In $S1(initialize)$, z, O and AP are set to -1, $\{\}$ and $\{P_0\}$ respectively. The dependency graph of P_0 is that of the example dependency forest. This is represented by $rem = []$. *get_ub* sets the upper bound value (=63) of P_0 to UB. In practice, this is calculated by obtaining the maximum well-covered arc set of P_0. In $S2(search)$, *select_problem* selects P_0 and $get\_fs(P_0)$ is executed. The feasible solution FS and its score LB are calculated based on the algorithm in Fig.5 to set $FS = [14, 2, 16, 23, 29]$, $LB = 50$ (P_0 in the search diagram). $S3(incumbent\ value\ update)$ updates z and O to new values. Then, $get\_iapl(P_0)$ computes the inconsistent arc pair list $[(2, 15), (15, 23), (23, 18), (2, 18)]$ from the maximum well-covered arc set $[14, 2, 15, 23, 18]$ and set it to $IAPL$. $S5(maximum\ value\ test)$ compares the upper bound value UB and the feasible solution value LB. In this case, $LB < UB$ holds, so $BACL$ is assigned the value of $IAPL$. The next step $S6(branchoperation)$ executes the *graph_branch* function. *graph_branch* selects the arc pair with the highest arc score and performs the graph branch operation with the selected arc pair. The following is a $BACL$ shown with the arc names and arc scores.

$$[(nc2[17], pre15[10]), (pre15[10], sub23[10]),$$
$$(sub23[10], vpp18[9]), (nc2[17], vpp18[9])]$$

Scores are shown in [ ]. The arc pair containing the highest arc score is $(2, 15)$ and $(2, 18)$ containing $nc2[17]$. Here, $(2, 15)$ is selected and partial-problems $P_1(rem[2])$ and $P_2(rem[15])$ are generated. P_0 is removed from AP and the new two partial-problems are added to AP resulting in $AP = \{P_1, P_2\}$. Then, based on the best bound search strategy, $S2(search)$ is tried again. *select_problem* selects P_1 because the upper bound value of P_1 (=61) is greater than that of P_2 (=59). Since the upper bound of P_1 (=61) is greater than the feasible solution score (=51), *get_iapl* is executed and sets $BACL$ to the value shown in P_1 in Fig.7. The graph branch function *graph_branch* gets two candidates for child partial-problems corresponding to $rem[24, 2]$ and $rem[23, 2]$ because the inconsistent arc pair $(24, 23)$ is selected as the source of the graph branch operation (arc 24 has the highest score of 15). The former candidate for $rem[24, 2]$ is pruned immediately, because its upper bound value (=46) is smaller than the incumbent value (=51) (termination by the upper bound test). Therefore, *graph_branch* returns $\{P_3(rem[23, 2])\}$. The upper bound value UB of P_3 is 58 which is less than that of its parent problem P_1. The processing for P_1 is completed and P_1 is removed from AP. *select_problem* selects P_2 by comparing the upper bound values of P_2 and P_3 in AP. Partial-problem P_2 is terminated because it has no feasible solution ($FS = no\_solution$). Then, the next partial-problem P_3 is processed. P_3 has a feasible solution with a score of 41. Updating of the incumbent value does not occur because the obtained score is lower than the existing incumbent value. The next partial-problem P_4 has no feasible solution, so all processing is terminated at $S8(stop)$. At this time, the values of O and z are the optimum solution(=$\{[14, 24, 15, 31, 18]\}$) and its score (=51) respectively. This solution corresponds to the dependency tree (a) in Fig.**??**.



0,I : [i]-n-0
1,saw : [saw]-v-1
2,a : [a]-det-2
3,girl : [girl]-n-3
4,with : [with]-pre-4
5,a : [a]-det-5
6,telescope : [telescope]-n-6
7,in : [in]-pre-7
8,the : [the]-det-8
9,forest : [forest]-n-9
root : [root]-x-root

	33	4	6	14	16	11	12	29	26	27	23	24	42
33	—	○	○	○	○	○	○	○	○	○	○	○	○
4	○	—	○	○	○	○	○	○	○	○	○	○	○
6	○	○	—	○	○	○	○	○	○	○	○	○	○
14	○	○	○	—		○	○	○	○	○	○	○	○
16	○	○	○		—	○	○		○	○	○	○	○
11	○	○	○	○	○	—	○	○	○	○	○	○	○
12	○	○	○	○	○	○	—	○	○	○	○	○	○
29	○	○	○	○		○	○	—			○	○	○
26	○	○	○	○	○	○	○		—		○	○	○
27	○	○	○	○	○	○	○			—	○	○	○
23	○	○	○	○	○	○	○	○	○	○	—	○	○
24	○	○	○	○	○	○	○	○	○	○	○	—	○
42	○	○	○	○	○	○	○	○	○	○	○	○	—

Fig.8: DF for the example sentence including PP attachments

4.2 Prototypical Ambiguous Sentences

In addition to the previous example for homophone ambiguities, this section shows two examples of prototypical ambiguous sentences.

4.2.1 PP-attachment Ambiguity

Fig.8 shows a dependency forest for "I saw a girl with a telescope in the forest". There are no homophones in the forest but two prepositional phrases with attachment ambiguities. The preposition "with" has two possible dependencies (*npp*14,*vpp*16) and "in" has three (*vpp*27,*npp*26,*npp*29). The combination number of these arcs is $2 * 3 = 6$, but there exists five well-formed dependency trees due to the existence of the co-occurrence constraint between arcs 16 and 29 ($CM(16, 29) \neq \bigcirc$) corresponding to the no crossing arc constraint. The scores of these arcs are

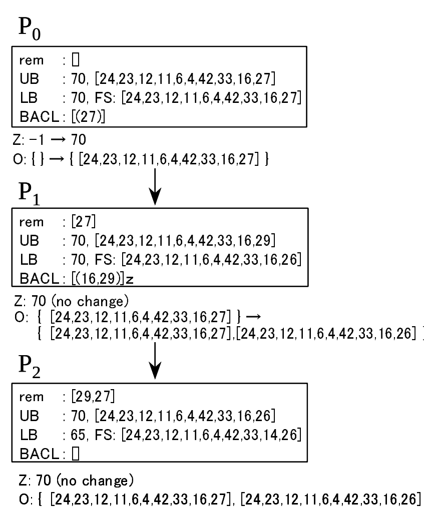


Fig.9: Search diagram for the example sentence including PP attachments

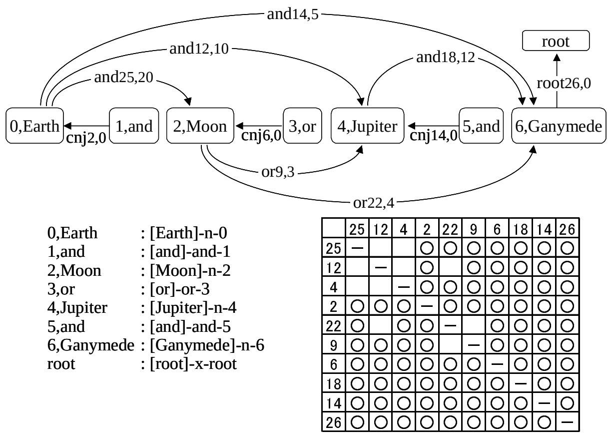


Fig.10: DF for the example sentence including coordinates

assumed to be calculated based on the preposition, the governor and dependant nodes of the preposition. *vpp*16 has a higher score compared with *npp*14 because "telescope" is a tool for seeing something. On the other hand, *vpp*27,*npp*26 and *npp*29 have the same scores. The search diagram for this example is shown in Fig.9. P_0 generates the optimum solution ($UB = LB$) with a score of 70. $S8$(*search more optimum solution*) in Fig.4 is executed. P_0 has no graph branch candidates in the inconsistent arc pair list ($IAPL == \{\}$). $arcs\_with\_alternatives(FS)$ selects arc *vpp*27 as a candidate of graph branching because it has rival arcs with the same score (*npp*26,*npp*29). Then P_1 is generated to obtain the second optimum solution including *npp*26. Next P_2 with $rem[26, 27]$ is generated and a feasible solution to P_2 is calculated. This solution is not added to the incumbent solution list because it has a lower score (65) than the obtained optimum solutions. This example has two optimum solutions.

4.2.2 Coordination Scope Ambiguity

Fig.10 shows a dependency forest for "Earth and Moon or Jupitor and Gamymede". Corresponding to the combination of the scopes of the three coordinations, "Earth" and "Moon" have three and two outgoing arcs, respectively. Since there exists a co-occurrence constraint (no crossing arc constraint) between *and*12 and *or*22, the dependency forest has five well-formed dependency trees. Arc scored are assigned assuming preference knowledge like "Planet names tend to co-occur" and "The name of a planet and its secondary planet tend to co-occur".

The search diagram for this example is shown in Fig.11. The feasible solution to the initial problem P_0 happens to be the optimum solution. No branch operation is performed because $IAPL$ of P_0 is [] and all arcs in the optimum solution have no rival arcs.

5 Dynamic Programming and Branch and Bound Method

Ref.10) described related work in the graph branch method and mentioned some researches on optimum tree search algorithms based on the dynamic programming (DP) framework. This section describes why PDG has not adopted some DP-based algorithm but rather the graph branch method based on the branch and bound framework for the optimum tree search.

Ref.5) proposed an algorithm for obtaining the optimum kakari-uke tree and its score from a set of all possible scored kakari-uke relations. This algorithm can be extended to treat general dependeny relations[6]. This algorithm is generalized into the minimum cost partitioning method (MCPM) which is a partitioning computation based on the recurrence equation given below[14]. MCPM is also a generalization of the probabilistic CKY algorithm and the Viterbi algorithm [*9].

Considering the phrase $(w_i, \cdots w_j; a_i, \cdots, a_j; A)$ partitioned into $(w_i, \cdots, w_k; a_i, \cdots, a_k; B)$ and $(w_{k+1}, \cdots, w_j; a_{k+1}, \cdots, a_j : C)$ where w_x, a_x, and A-C mean word, analog information (like prosodic information), and features like phrase name, respectively. MCPM computes the optimum solution based on the following recurrence equation for total cost F.

$$F(i, j, A) = min[F(i, k, B) + F(k + 1, j, C) + cost(w_i, \cdots, w_j, a_i, \cdots, a_j, k, A, B, C)]$$

$F(i, j, A)$ is the total cost of phrase A covering from the i-th to the j-th word in a given sentence. $cost(w_i, ...w_j, a_i, ..., a_j, k, A, B, C)$ is a cost function where k is a partitioning position. The minimum cost partition of the whole sentence is calculated very efficiently by the DP principle for this equation. The optimum partitioning obtained by this method constitutes a tree covering the whole sentence satisfying the single role and no cross dependency constraints. However, the single valence occupation constraint adopted in PDG for basic semantic level constraint is not assured to be satisfied by MCPM.

Fig.12 shows a dependency graph for the Japanese phrase "Isha-mo wakaranai byouki-no kanjya" encompassing dependency trees corresponding to "a patient suffering from a disease that the doctor doesn't know", "a sick patient who does not know the doctor", and so on. The dependency graph has two kinds of ambiguities, i.e. semantic role ambiguity and attachment ambiguity. For example, $wakaranai(not\_know)$ has four outgoing arcs with different semantic roles (*agent* and *target*) and different attachments ($byouki(sickness)$ and $kanjya(patient)$) as shown

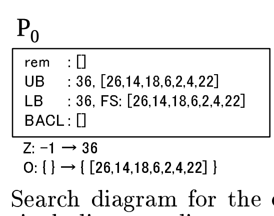


Fig.11: Search diagram for the example sentence including coordinates

*9 Specifically, MTCM corresponds to probabilistic CKY and the Viterbi algorithm because it computes both the optimum tree score and its structure.

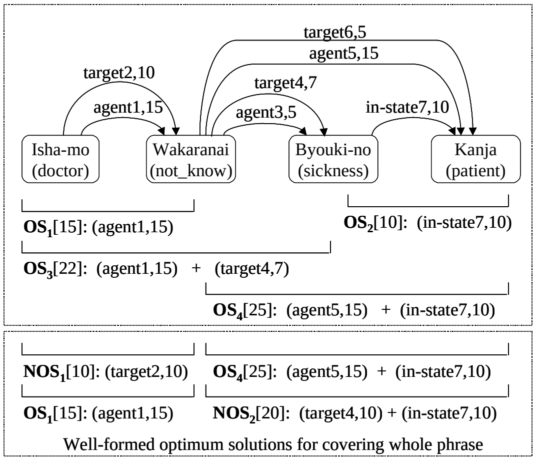


Fig.12: Optimum solution search satisfying the single valence occupation constraint

in Fig.12. The single valence occupation constraint prevents $wakaranai(not\_know)$ from being connected with the same two semantic role arcs. OS_1 - OS_4 represent the optimum solutions for the phrases specified by their brackets computed based on MCPM. For example, OS_2 gives an optimum tree with a score of 22 (consisting of $agent1$ and $target4$) for the phrase "Isha-mo wakaranai byouki-no". The optimum solution for the whole phrase is either $OS_1 + OS_4$ or $OS_3 + OS_2$ due to MCPM. The former has the highest score $40(= 15 + 25)$ but does not satisfy the single valence occupation constraint because it has $agent1$ and $agent5$ simultaneously. The optimum solutions satisfying this constraint are $NOS_1 + OS_4$ and $OS_1 + NOS_2$ as shown at the bottom of Fig.12. NOS_1 and NOS_2 are non optimum solutions for their word coverages. In this case, MCPM generates a non-optimum tree in $OS_3 + OS_2$ if it adopts the strategy of neglecting inconsistent trees. Otherwise, MCPM generates an high score but an ill-formed tree in $OS_1 + OS_4$. This shows that MCPM is not assured to obtain the optimum solution satisfying the single valence occupation constraint. On the contrary, the graph branch algorithm is assured to compute the optimum solution(s) satisfying any co-occurrence constraints in the co-occurrence matrix including the single valence occupation constraint. It is an open problem whether there exists an algorithm based on the DP framework which can handle the single valence occupation constraint and arbitral arc co-occurrence constraints.

6 Concluding Remarks

This paper has described the graph branch algorithm for obtaining the optimum solution for a dependency forest used in the preference dependency grammar. The proposed graph branch algorithm has wider applicability compared with the semantic dependency graph because it can handle whole morphological ambiguity caused by homonyms and word boundary divisions. The advantage of the graph branch method compared with the minimizing total cost method is that it can handle arbitral arc co-occurrence constraints including the single valence occupation constraint, which is a basic semantic-level constraint in preference dependency grammar.

Refer to Ref.10) for a discussion of related work, the computational complexity and some optimization techniques for the graph branch algorithm.

References

[1] McDonald,R., Crammer,K and Pereira,F: *Spanning Tree Methods for Discriminative Training of Dependency Parsers* , UPenn CIS Technical Report MS-CIS-05-11, (2005)
[2] Collins, M.: *Head-Driven Statistical Models for Natural Language Parsing*, Ph.H. thesis, University of Pennsylvania, (1999)
[3] Charniak, E.: *A manimum-entropy-inspired parser* Proceedings of NAACL, (2000)
[4] Nivre, J. and Scholz, M.: *Deterministic Dependency Parsing of English Text* Proceedings of COLING'04, (2004)
[5] Ozeki,K.: *Dependency Structure Analysis as Combinatorial Optimization*, Information Sciences 78(1-2), 77-99, (1994)
[6] Katoh, N. and Ehara, T.: *A fast algorithm for dependency structure analysis* (in Japanese), Proceedings, 39th Annual Convention of the Information Processing Society, (1989)
[7] Eisner,J: *Three new probabilistic models for dependency parsing: An exploration* Proceedings of COLING'96, (1996)
[8] Yamada, H. and Matsumoto, Y.: *Statistical dependency analysis with support vector machine* Proceedings of IWPT'03, (2003)
[9] Hirakawa. H. and Amano, S.: *Japanese sentence analysis using syntactic/semantic preference* (in Japanese), Proceedings of the 3rd National Conference of JSAI, pp. 363-366, (1989)
[10] Hirakawa, H.: *Semantic dependency analysis method for Japanese based on optimum tree search algorithm*, Proceedings of the PACLING 2001, (2001)
[11] Hirakawa, H.: *Dependency Forest: Packed Shared Dependency Structure Corresponding to Parse Forest* (in Japanese), IPSJ, Natural Language Processing NL-167-9, (2005)
[12] Ibaraki,T.: *Branch-and-bounding procedure and state-space representation of combinatorial optimization problems*, Information and Control,36,1-27,(1978)
[13] Tomita, M.: *Generalized LR Parsing* , Kluwer Academic Publishers, Boston, MA, (1991)
[14] 尾関和彦, 張玉潔, "最小コスト分割問題としての係り受け解析," 言語処理学会第 5 回年次大会ワークショップ論文集, pp.9 -14 (1999)