大規模コーパス作成支援システム Bonsai

吉田 恭祐† 橋本 泰一† 徳永 健伸† 田中 穂積†

† 東京工業大学大学院 情報理工学研究科 計算工学専攻

{rincho, taiichi, take, tanaka}@cl.cs.titech.ac.jp

1 背景と目的

近年，自然言語処理の分野では，大規模な言語資源に基づく統計的手法が研究の中心となっている．そのため，大規模な言語資源の作成が必要となるが，それを全て人手により行うのは，困難である．この問題を解決するために，言語資源作成支援システムに関する研究が盛んに行われている [6, 1, 5]．

過去の事例を参考にしながら言語資源を作成することができれば，言語資源を一貫性を保ちながら作成することが可能になる．我々は，言語資源作成支援システムとして，コーパスに構文構造を付与するためのツール Bonsai を構築している．このツールは，コーパス作成中に文法の一貫性を確認するための検索機能を提供しており，ユーザは検索したい構文構造をクエリとして与えることで，その構造を含む文を構文構造付きコーパスから検索できる．

本論文では，この検索機能の仕組みについて述べる．まず，2 節で構文構造付きコーパスを関係データベースに変換する方法について述べ，3 節で構築したデータベースの検索手法について述べる．検索には SQL を用いるが，検索条件の絞り込みの工夫，最適な SQL 文の作成により，検索時間を高速化する手法を提案する．

2 コーパスのデータベース化

我々は木構造をデータベース化する方法として，XML に着目した．吉川らは XML 文書をオブジェクト関係データベースを用いて効率良く格納，検索する手法を提案している [3, 4]．この手法では「出現位置」というデータを用いて，ノードが木の構造のどの部分に位置するかをうまく表現している．我々は吉川らの手法を参考に，コーパスのデータベース化を以下のように実現した．

本論文では，Penn Treebank コーパス [2]48,884 文をデータベース化した．各文は，図 1 のような構文木で表現される．データベース化は，各ノード毎に表 1 のような情報を基本テーブルに格納することにより

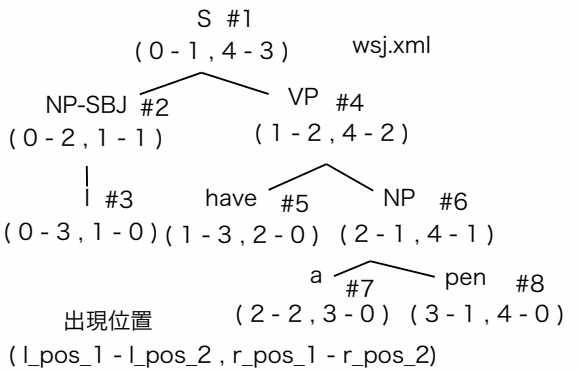

図 1: 構文木の例

行なう[1]．sentenceID,nameID,pathID は正数で与え，それらに対応する文字列はそれぞれ SentenceTable, NameTable, PathTable に格納する．

フィールド名	格納する値
sentenceID	その構文木の名前の id (wsj.xml に対する id)
nodeID	ノード毎にユニークな数値．先行順で与える．(♯5)
nameID	そのノードのタグ名の id (”have”に対する id)
pathID	ルートからのパスの id (”S/VP/have”に対する id)
parentID	そのノードの親の nodeID (♯4)
nextSibID	そのノードの弟の nodeID (♯6)
l_pos_1,l_pos_2	出現位置．4 つの正数で与える．
r_pos_1,r_pos_2	(順に 1,3,1,0)

表 1: 基本テーブルの概要

「出現位置」は図 1 のように 4 つの正数で与える．このような与え方により図 2 のようなノード間の位置関係が表現できる．包含関係は，X の子孫ノードが X ノードをルートとする木の中のノードであることを特定する．親戚関係は，i ノードと j ノードの左右に関する位置を特定する．

3 検索手法

3.1 検索の概要

クエリは図 3 左のような木構造で与える．∗ はワイルドカードで，任意の label を許す．図 3 のクエリの場合，図 1 のようにクエリを部分木としてもつ構文木が検索結果となる (この場合，♯4, ♯5, ♯6 の部分とマッ

[1]各フィールドの説明の後に，図 1 のノード”have”に対する具体例を ( ) 内に示す．

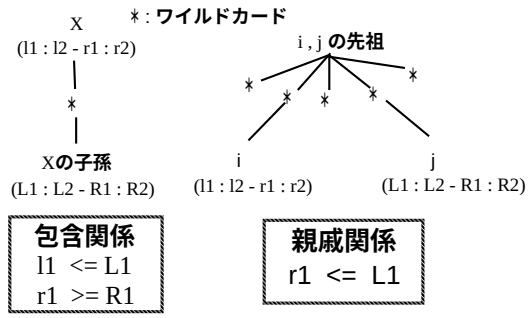

図 2: 包含関係と親戚関係

チする)．ただし，図 4 に示す Exact Match と Partial Match のどちらを指定するかによって，検索結果は異なる．Exact Match はクエリの子供の数と該当する部分の子供の数が必ず等しくなければならないが，Partial Match は，部分的にクエリと構造が等しければ，答えとして出力する．本システムでは，ユーザの検索要求に合わせて，どちらか選択できるようになっている．

図 3 左のクエリは同図右の SQL 文で検索する．このように，SQL 文では，クエリのノード数に応じて from 節でそれぞれ宣言し，where 節で各ノードに対する条件を書く．従って，クエリのノード数が多くなれば，それに応じて SQL 文も長くなる．3.2 節で行った実験により，ノード数が 18 以上では，検索時間が大幅に遅くなることが分かった．したがって，クエリをそのまま SQL1 文で検索するという手法では，ノード数の大きいクエリに対応できない．この問題を解決するため，クエリを適当な部分木に分割して，それぞれを別の SQL 文で検索するという手法を提案する．

3.2 検索単位

どのくらいの大きさの部分木が SQL1 文に適しているのか調べるため，クエリのノード数に対する検索時間の比較実験を行った．実験に用いたクエリは，Penn Treebank コーパス中の 4 文から抜き出した全ての部分木である．検索手法としては，図 4 の 2 つが考えられるが，実験では，どちらの検索手法でも安定する部分木を見つけるため，条件が緩く検索時間がより大きい Partial Match を用いた．その実験結果を表 2 に示す．

平均検索時間からすると，ノード数が 17 から 18 で検索時間が大幅に増えているのが分かる．これより，部分木のノード数は 17 以下にするべきだと考えられる．また，ノード数が 12 以内の場合は，最大検索時

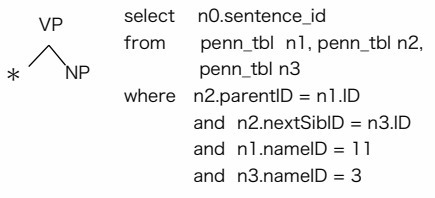

図 3: クエリと SQL 文の例

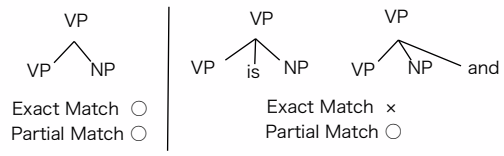

図 4: 図 3 をクエリとした場合の例

間からも分かるように，部分木に分割せず，SQL1 文で検索したほうが早いことが分かった．

次に，同じ実験結果を深さ別に分類したものを表 3 に示す．深さ 7 で検索時間が大きくなっているのは，ノード数の多さが原因だと思われる．また，深さ 1 の検索時間も大きくなっているが，これは深さ 1 では条件が少なすぎて，出力数が多くなっていることが原因だと考えられる．絞り込みをうまく行うには，個々の部分木の出力数はなるべく少ないほうが望ましいので，深さ 1 は適してないと言える．以上のことをふまえて，以下のような検索戦略を用いる．

1: ノード数が 12 以内のクエリは分割しない
2: 基本的に，深さ 2 で分割
3: ノード数が 18 以上の場合は，更に 2 分割

この戦略に基づき，クエリから逐次的に部分木へ分割を行う．ここで，分割される各部分木のことを検索単位と呼ぶことにする．深さを 2 としたのは，深さ 3 ではこの逐次処理において頻繁にノード数が 18 を越えてしまい，再分割のコストがかかってしまうからである．また，18 という閾値は最小検索時間から決定した．表 2 の最大検索時間を考慮すると，各検索単位のノード数を常に 12 以内としたほうが良いように思われる．しかし，これらの最大検索時間は SQL 文の書き方を工夫することで改善することが可能である．この SQL 文の最適化については，5 節で詳しく述べる．

3.3 検索単位の順序

検索をするにあたり，どの検索単位を最初に検索するかは検索効率に大きく影響する．最初の検索単位とマッチする数が少なければ，候補の数を早く絞り込む

ノード数	平均検索時間 (sec)	最大検索時間	最小検索時間	平均出力数 (部分木)	実験したクエリ数
2	2.04	10.63	0.001	24330.30	65
3	1.59	11.33	0.001	9854.96	99
4	0.84	11.51	0.002	2189.00	168
5	0.38	10.05	0.002	337.26	281
6	0.19	6.42	0.002	65.11	437
7	0.12	1.64	0.003	17.00	611
8	0.09	1.64	0.003	7.16	762
9	0.07	1.35	0.004	3.32	860
10	0.06	1.31	0.008	1.91	902
11	0.05	1.31	0.009	1.36	901
12	0.07	1.32	0.010	1.13	869
13	0.11	3.84	0.012	1.04	801
14	0.18	9.36	0.018	1.01	678
15	0.28	45.45	0.041	1.00	498
16	0.56	2.25	0.065	1.00	300
17	1.94	6.12	0.270	1.00	140
18	7.51	18.09	1.093	1.00	47
19	31.13	47.52	8.021	1.00	10
20	161.02	185.45	135.250	1.00	3
21〜	1108.17	3394.60	298.33	1.00	23

表 2: ノード数別平均検索時間

深さ	平均検索時間 (sec)	最大検索時間	最小検索時間	平均出力数 (部分木)	実験したクエリ数
1	2.08	10.63	0.001	19092.0	134
2	0.34	11.33	0.001	609.0	694
3	0.14	45.45	0.002	24.6	2544
4	0.16	1.64	0.007	38.6	458
5	0.13	1.35	0.007	8.3	760
6	0.16	18.09	0.007	2.0	1680
7	2.60	3394.60	0.020	1.0	2185

表 3: 深さ別平均検索時間

ことができ，検索時間は短くなる．つまり，最初に検索する検索単位の候補数が最も少なくなるように，クエリを分割することが重要となる．これを実現するため，次のように最初に検索する検索単位を決定する．

図 6 をクエリとした場合を例として説明する．2 節で述べたように，nameID の指し示す label(”NP”や”VP”など) は，図 5(1) のような NameTable に格納される．NameTable には，label ごとの頻度が frequency フィールドに格納してある．また，図 5(2) のような ChildrenTable には，子ノードの label のリストを値に持つ children フィールドがあり，それぞれの頻度が frequency フィールドに格納してある．図 5(2) の最上部の行の場合，親ノードの label が S で，1 番目の子ノードの label が NP-SBJ, 2 番目が VP であるノードの頻度が 41,604 であることを示す．この 2 つのテーブルより，クエリの全てのノードに頻度を与える．

このように，全てのノードに頻度を与えた後，その中で最小の頻度を持つノードとその子ノードより構成される部分木が最初の検索単位に含まれるようにする．図 6 の例では，NP-SBJ が頻度最小のノードであるので，その子ノード Indian, NP より構成される部

name	nameID	frequency
S	1	106901
NP-SBJ	2	94319
VP	11	179161
Indian	12501	36
movie	1820	74
star	2971	58

(1) Name Table

parent	children	frequency
S	NP-SBJ:VP	41604
NP-SBJ	Indian:NP	13
NP	movie:star	25

(2) Children Table

図 5: 頻度決定に用いるテーブル

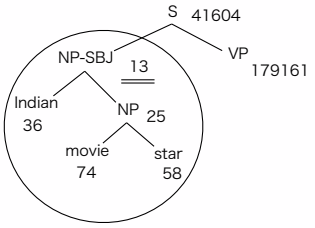

図 6: 最初の検索単位の決定方法

分木がその対象となる．次に，3.2 節で述べたように，検索単位は深さ 2 を基本とするので，この部分木を深さ 1 だけ拡張する．最小頻度ノードの親ノードの頻度が，最小頻度ノードの全ての子ノードの頻度よりも小さい場合は上へ，その他の場合は下へと拡張する．図 6 の例では，下へと拡張されており，深さ 2 の最初の検索単位が決定されている[2]．

最初に検索する検索単位決定後，3.2 節で述べた定義に基づき，クエリの残りの部分を逐次的に，検索単位に分割する．

最初の検索単位の次に検索する検索単位は，隣接する検索単位の中から選ぶ．その際，なるべく候補を絞れる検索単位を選ぶのが望ましい．そこで，隣接する検索単位の中で最小の頻度ノードを含む検索単位を次に検索する．それ以降も同様の基準で検索順を決める．

4 実験

検索時間についての実験を行った．実験に用いたクエリは，Penn Treebank コーパス中の 1,924 文から抜き出した全ての部分木 22,402 個からランダムに 463 個選んだものである．検索手法としては，Exact Match を用いた．また，検索するデータベースには，Penn Treebank コーパス 48,884 文が格納されてある．

まず始めに，クエリを常に SQL1 文で検索するものとの比較を行った．ここで，提案手法ではノード数が 12 以下のものはクエリの分割を行わず同じ検索手法

[2]便宜上，ノード数の少ないクエリで説明したが，このクエリはノード総数が 12 以下なので，厳密には検索単位には分割しない．

となるので，実験対象とはしない．表 4 にその結果を示す．提案手法では，ノード数が増えても平均検索時間の差は見られないが，SQL1 文で検索する手法では，ノード数が 17 以上で特に検索時間が大きくなっているのが分かる．

次に，最初に検索する検索単位が検索時間をどれだけ短縮させているかを調べるため，提案手法と同様に検索単位を分割した後，トップダウンに検索単位を検索するものとの比較を行った．この実験でも，ノード数が 12 以下のクエリは実験対象としていない．表 4 にその結果を示す．

全体的に見て，トップダウン手法の約 6 分 1 の時間で検索していることが分かる．この結果より，最初の検索単位での検索候補の絞り込みがうまくいっていると言える．

ノード数	クエリ数	提案手法	SQL1 文	トップダウン
13	50	0.039	0.151	0.383
14	49	0.045	0.174	0.318
15	49	0.047	0.386	0.226
16	36	0.024	1.972	0.555
17	31	0.280	12.671	0.991
18	36	0.033	9.797	0.360
19	32	0.042	41.448	0.522
20	29	0.043	75.802	0.048
21	19	0.036	298.331	0.039
22	22	0.092	787.267	0.667
23	25	0.041	723.551	0.173
24	39	0.048	1369.466	0.942
25	46	0.046	3394.602	0.094

表 4: 検索時間の比較

5 SQL 文の最適化

3.2 節で述べたように，SQL 文の書き方を工夫することで，検索時間を短縮できる．表 2 でノード数が 13 以上のクエリでは，検索時間の最大，最小の差が大きくなっている．これらのクエリを調べてみたところ，クエリの出力数，深さには特に原因は見られなかった．そこで，最大検索時間となったいくつかのクエリについて，本検索システムが作成した SQL 文の where 節の順序を人手でいくつか入れ替えてみたところ，全てのクエリにおいて，検索時間を短縮するような順序を発見できた．つまり，SQL 文の順序によって，検索時間が大きく変わってしまう．

本検索システムでは，SQL 文の作成過程において，各検索単位の素性を用いてはいない．しかし，今回の調査より，検索単位の素性によって，検索時間が早くなるような SQL 文の順序があることが分かった．ノード数，深さ，3.3 節で求めた頻度情報などの素性を SQL 文の順序に用いることで，検索時間を更に短縮できる可能性がある．また，検索時間の遅くなるような順序を求めることにより，安定したシステムの構築が可能であろう．今回の実験では，どのような書き順で SQL 文を書けばよいか，はっきりとは分からなかったが，これについては，十分改良の余地がある．

6 まとめと今後の課題

本論文では，吉川の手法を基に関係データベースに格納した構文構造付きコーパスから構文木を検索する手法の改善方法を提案した．Penn Treebank における SQL1 文に適した部分木 (検索単位) を実験的に求めた．更に，候補数の少ないものから検索できるようにクエリを分割する手法を提案した．この手法により，実験ではほぼ全てのクエリに対して 0.1 秒以内で検索結果を出力することができた．

今後の課題として，以下のようなものが挙げられる．

- label を指定しないクエリによる提案手法の評価実験
- クエリと類似した部分木を持つ文を検索する手法の提案
- Bonsai システムのユーザインタフェースの改善
- コーパスの修正・作成機能，コーパスの一貫性を自動的に管理する機能などの機能拡張

参考文献

[1] H. Cunningham, V. Tablan, K. Bontcheva, and M. Dimitrov. Language engineering tools for collaborative corpus annotation. In *Proceedings of Corpus Linguistics 2003*, 2003.

[2] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, Vol. 19, No. 2, 1993.

[3] 吉川正俊, 志村壮是, 植村俊亮. オブジェクト関係データベースを用いた XML 文書の格納と検索. 情報処理学会論文誌, Vol. 40, No. 0, 1999.

[4] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. Xrel: A path-based approach to storage and retrieval of xml documents using relational database. *ACM Transactions on Internet Technology*, Vol. 1, No. 1, 2001.

[5] Oliver Plaehn and Thorsten Brants. Annotate – an efficient interactive annotation tool. In *Proceedings of the Sixth Conference on Applied Natural Language Processing ANLP-2000*, Seattle, WA, 2000.

[6] 工藤拓, 松本裕治. RDB を利用したタグ付きコーパス検索支援環境の構築. 情報処理学会研究報告 NL-144, pp. 135–142, 7 2001.