임의 순서 차트 파싱 알고리즘
(Random Order Chart Parsing Algorithm)

심 광 섭 †
(Kwangseob Shim)

요 약 차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행하여야 한다는 제약이 따른다. 본 논문에서는 이러한 제약을 없앤 임의 순서 차트 파싱 알고리즘을 제안한다. 제안한 알고리즘에서는 입력 문장의 각 단어에 대하여 어떤 순서로 파싱을 하더라도 무방하다. 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행하는 것은 임의 순서로 파싱을 진행하는 것의 특수한 형태이므로 임의 순서 차트 파싱 알고리즘에서도 입력 문장의 왼쪽에서 오른쪽으로 파싱을 하는 것이 가능하다. 제안된 알고리즘은 차트 파싱 알고리즘을 확장한 것으로서 제어 구조가 매우 단순하며 구현도 용이하다.

키워드 : 차트 파싱, 섬 파싱, 임의 순서 차트 파싱

Abstract According to the original chart parsing algorithm, a sentence is parsed in a strict left-to-right order. The modified chart parsing algorithm proposed in this paper breaks the strictness. With the proposed algorithm, a sentence is parsed in a random order. Conventional left-to-right parsing is also possible, since left-to-right parsing is a special case of random-order parsing. The proposed parsing algorithm is an extension of chart parsing algorithm and its control structure is very simple, so that it is easy to implement the algorithm.

Key words : chart parsing, island parsing, random order chart parsing

1. 서 론

대표적인 파싱 알고리즘 중의 하나인 차트 파싱(chart parsing) 알고리즘에서 파싱 시간은 문장 길이의 세제곱에 비례한다[1]. 따라서 문장이 길면 길수록 파싱 시간이 급격하게 늘어날 뿐만 아니라 파싱에 필요한 메모리의 양도 폭발적으로 증가한다. 메모리 공간의 크기가 한정되어 있으므로 긴 문장의 경우 파싱 도중에 메모리 공간이 다 소진되어 중도에 파싱을 포기해야 하는 상황이 발생할 수도 있다. 차트 파싱에서는 주어진 문장의 왼쪽에서 오른쪽으로 순차적으로 진행하면서 파싱을 수행하기 때문에 중도에 파싱을 포기한다는 것은 파스 트리를 하나도 생성하지 못한다는 것을 의미한다. 자연어 처리 시스템에서 이러한 상황이 발생하는 것은 별로 반가운 일이 아니다. 예를 들어, 기계 번역 시스템에서 이러한 상황이 발생한다면 번역문의 질은 고사하고 아예 번역문을 생성하지도 못하게 되는 것이다. 그런데 최근에는 웹문서를 처리 대상으로 하면서 긴 문장에 대한 파싱 성공률을 높이는 것이 점차 중요시되고 있다.

긴 문장에 대한 파싱을 어렵게 하는 요인 중에는 모호성(ambiguity) 문제가 있다. 하나의 단어가 여러 품사로 사용될 수 있다는 품사 모호성도 그 중의 하나인데, 이러한 모호성을 그대로 안고 갈 경우 파싱 단계에서 파스 트리(parse tree)가 폭발적으로 증가할 수 있다. 이러한 문제를 피하기 위하여 파싱 전에 품사 태거(POS tagger)를 두는 경우가 많다. 실제로 [2]에서는 파스 트리의 생성을 최대한 억제하기 위하여 주어진 문장에서 가장 적절하다고 판단되는 품사열 하나를 선택하고 이것만으로 파싱을 하는 방식을 취하고 있다. 그런데 현재 가장 우수하다는 품사 태거의 정확도는 97% 정도로, 이는 약 32 단어마다 한 단어의 비율로 품사 태깅 오류가 발생함을 의미한다. 따라서 32 단어가 넘는 긴 문장은 평균적으로 하나 이상의 품사 태깅 오류가 포함될 수 있으며, 이러한 오류는 파싱 실패의 주된 원인 중의 하나가 된다.

† 종신회원 : 성신여자대학교 IT학부 교수
shim@sungshin.ac.kr
논문접수 : 2009년 8월 26일
심사완료 : 2010년 4월 11일

정보과학회논문지: 소프트웨어 및 응용 제37권 제6호(2010.6)

품사 태거의 정확도를 높이는 데에는 한계가 있으므로 한 단어에 대하여 하나의 최적 품사를 부여하지 않고 복수 품사를 부여하여 파싱을 하는 방법을 취할 수도 있다. 하지만 품사 태거의 정확도를 보완하기 위하여 한 단어를 여러 품사로 태깅했다 하더라도 가장 가능성이 높은 품사를 가지고 파싱을 먼저 해 보고 파싱에 실패한 경우에 한하여 다음으로 가능성이 높은 품사를 가지고 파싱을 할 수 있다면 파싱 실패 문제에 대하여 보다 유연하게 대처하면서도 파스 트리 생성을 억제하는 효과도 거둘 수 있을 것이다. 그러나 기존 차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행해야 하므로 이러한 방식을 취하기가 곤란하다.

본 논문에서는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행해야 한다는 기존 차트 파싱 알고리즘의 제약 조건을 제거하여 임의 순서로 파싱을 할 수 있도록 수정한 차트 파싱 알고리즘을 제안한다. 임의 순서로 파싱을 할 수 있기 때문에 일단 가능성이 가장 높은 품사열을 가지고 파싱을 해 보고 파싱에 실패한 경우 그 다음으로 가능성이 높은 품사를 점진적으로 추가해 가면서 파싱을 진행하는 것이 가능하다.

2. 관련 연구

입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행해야 한다는 제약 조건을 제거한 알고리즘으로 섬 파싱(island parsing) 알고리즘이 있다[3]. 이 알고리즘은 음성 인식 등에 활용하기 위하여 제안된 것으로[3,4], 입력 문장에서 하나 이상의 섬(island)을 지정하고 이 섬의 좌우로 파싱을 진행해 나간다는 개념이다. [3]에서 섬을 중심으로 양방향으로 파싱을 진행하는 방식을 취한 이유는 음성 인식 결과 어떤 부분은 명확하게 인식할 수 있는 반면 다른 부분은 명확하게 인식을 할 수 없기 때문에 문장의 왼쪽 끝에서 시작하여 오른쪽으로 단방향 파싱을 하는 것보다는 명확하게 인식된 부분을 중심으로 좌우 양방향 파싱을 해 나가는 것이 타당하다고 보았기 때문이다.

차트 파싱 알고리즘에서는 문법 규칙 $A \rightarrow XYZ$의 오른쪽에 있는 성분 (constituent) X, Y, Z가 순서대로 인식되면 새로운 성분 A가 생성된다. 때문에 차트 파싱에서는 제어 구조가 간단하여 문법 규칙의 적용 과정을 관리하기가 비교적 용이하다. 반면, 섬 파싱 알고리즘에서는 문법 규칙의 임의 지점에서 시작하여 양방향으로 인식해 나가는데, 예를 들어 X를 인식하기 전에 Y가 먼저 인식되었다면 Y를 중심으로 양방향으로 파싱을 진행하면서 X와 Z를 인식하는 방식으로 파싱이 진행된다. 따라서 섬 파싱은 차트 파싱에 비하여 문법 규칙의 적용 과정을 관리하기가 까다로운 편이다. 뿐만 아니라 새로운 활성 아크(active arc)가 생성될 때마다 기존 활성 아크와 중복되는지 검사해야 하고 또 인접한 활성 아크가 있다면 이 둘을 하나로 통합하는 과정을 거쳐야 한다는 비효율성이 존재한다[5,6].

기존의 섬 파싱 알고리즘에서는 입력으로 단어열이 주어지는 것을 전제로 하고 있는데, [6]에서는 음성 처리 분야에 보다 적합한 입력 구조인 단어 그래프(word graph)를 처리할 수 있도록 수정한 알고리즘을 제안하였다. [7]에서는 지역적 양방향 파싱이라는 알고리즘을 제안하였는데 기본적으로는 섬 파싱 알고리즘과 동일하다고 볼 수 있다.

[8]에서는 점진적인 양방향 파싱(bidirectional incremental parsing)을 제안하였는데 이는 의학 문서로부터 단백질과 단백질 사이의 상호 작용과 관련된 정보를 추출하기 위한 목적으로 제안된 것으로 일반적인 의미의 파싱 알고리즘이라고 할 수는 없다. 여기서 제안된 방법에 의하면, 주어진 문장을 스캔하여 미리 정해진 동사 목록에 포함된 동사가 있는지 검사를 하는데 만약 그러한 동사가 발견된다면 그 동사를 중심으로 양방향으로 스캔을 하면서 정규 문법으로 정의된 패턴을 적용하여 명사구(NP) 후보를 찾는다. 명사구 후보는 CKY 기반의 파서를 사용하여 실제로 명사구가 될 수 있는지의 여부를 판단한다. 따라서 [8]에서 제안된 양방향 파싱은 문장 전체에 대한 파싱을 하는 것이 아닐 뿐만 아니라, 명사구 후보가 실제로 명사구인지의 여부를 판단하기 위하여 CKY 알고리즘을 사용하는 것으로 사실상 차트 파싱과 다를 바 없다.

3. 임의 순서 차트 파싱 알고리즘

3.1 차트 파싱 알고리즘

차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행하면서 입력 문장에서 주어진 혹은 파싱 과정에서 생성된 성분(constituent)을 가지고 문법 규칙을 점진적으로 적용해 나간다. 이처럼 차트 파싱에서는 문법 규칙을 점진적으로 적용하여 파싱을 하기 때문에 파싱 중에는 적용이 완료되지 않은 문법 규칙이 존재하게 된다. 예를 들어 문법 규칙 $A \rightarrow XYZ$가 있다고 하자. 이 문법 규칙은 성분 X, Y, Z가 차례로 주어졌을 때 이들을 자식으로 하는 새로운 성분 A를 생성하는 규칙이다. 만약 성분 X가 주어지면 이 문법 규칙의 적용을 시작할 수는 있으나 적용이 완전히 끝난 상태는 아니다. 이처럼 적용 중인 상태에 있는 문법 규칙을 활성 아크(active arc)라고 하며 $A \rightarrow X \cdot YZ$와 같이 나타낸다. 활성 아크에서 $\cdot$의 왼쪽에는 이미 인식된 성분들이 오며, 오른쪽에는 앞으로 인식해야 할 성분들이 온다. 성분 Y가 주어지면 활성 아크 $A \rightarrow X \cdot YZ$

는 활성 아크 $A \rightarrow XY \cdot Z$로 확장(extend)된다.

하나의 문법 규칙이 완전히 적용되면 새로운 성분이 생성되는데, 생성된 성분은 일단 어젠더(agenda)에 저장된다. 어젠더는 파싱 중에 생성된 성분 중에서 아직 문법 규칙 적용에 사용되지 않은 것들을 임시로 저장해 두는 곳이다[9,10]. 어젠더가 비어 있지 않다면 새로운 성분을 하나씩 꺼내 확장 가능한 모든 활성 아크를 확장하고, 적용 가능한 모든 문법 규칙으로부터 새로운 활성 아크를 생성하는 데 사용된다. 어젠더에서 꺼내어 활성 아크의 확장에 사용된 성분은 차트(chart)에 저장된다.

차트 파싱 알고리즘은 그림 1과 같이 설명할 수 있다. 여기서 대문자는 성분을 나타내며, 그리스 문자는 하나 이상의 성분의 열(sequence)을 나타낸다. 또한 $<A \rightarrow X \cdot YZ, i, j>$는 입력 문장의 단어 i와 단어 j 사이 구간에서 성분 X까지 인식되었으며, 단어 j에서 시작하는 구간에서 성분 Y가 오기를 기다리는 활성 아크 $A \rightarrow X \cdot YZ$를 나타낸다. 그리고 $<A, i, j>$는 입력 문장의 단어 i와 단어 j 사이의 구간에서 문법 규칙 $A \rightarrow \alpha$가 적용되어 생성된 성분 A를 나타낸다.

3.2 문제점

그림 1의 차트 파싱 알고리즘에 의하면 어젠더에서 꺼낸 성분 $<X, p, q>$에 대하여 구간 p에서 대기 중인 활성 아크 중에서 성분 X가 오기를 기다리는 활성 아크 $<A \rightarrow \alpha \cdot XY\beta, o, p>$가 있다면 이 활성 아크는 새로운 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$로 확장된다. 뿐만 아니라 성분 $<X, p, q>$와 문법 규칙 $B \rightarrow X\delta$로부터 새로운 활성 아크 $<B \rightarrow X \cdot \delta, p, q>$가 생성된다. 파싱이 진행되면서 $<X, p, q>$의 오른쪽에 위치한 인접 성분 $<Y, q, r>$이 주어지면 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$는 $<A \rightarrow \alpha XY \cdot \beta, o, r>$로 확장되며, 문법 규칙 $C \rightarrow Y\gamma$로부터 새로운 활성 아크 $<C \rightarrow Y \cdot \gamma, q, r>$이 생성된다. 이 과정을 그림으로 나타내면 그림 2와 같다.

만약 활성 아크의 확장에 사용될 성분들이 왼쪽에서 오른쪽으로 순차적으로 주어지지 않으면 그림 2와 같은

```
1 chartParsing(sentence of n words)
2 {
3    for i from 0 to n-1 do
4        add constituents < X, i, i+1 > to agenda
5    end
6
7    for any grammar rule of the form S→α do // assume S is a starting symbol
8        add an initial active arc < S→ · α, 0, 0 > to chart
9    end
10
11   while agenda is not empty do
12       given a constituent < X, p, q > ← pop(agenda) do
13           for all active arc of the form < A→α · X β, o, p > in chart do
14               if β is empty then
15                 add a new constituent < A, o, p > to agenda
16               else
17                 add a new active arc < A→αX · β, o, q > to chart
18               end
19           end
20           for a grammar rule B→Xδ do
21               if δ is empty then
22                 add a new constituent < B, p, q > to agenda
23               else
24                 add a new active arc < B→X · δ, p, q > to chart
25               end
26           end
27           add the constituent < X, p, q > to chart
28       end
29   end
30 }
```

그림 1 순차적 파싱에서의 활성 아크 확장

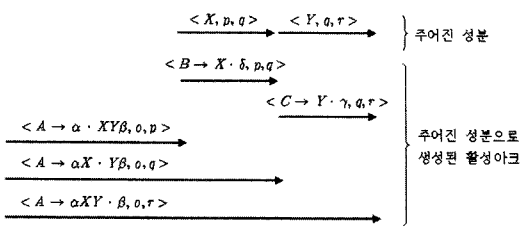


그림 2 순차적 파싱에서의 활성 아크 확장

결과를 얻지 못한다. 예를 들어 $p<q<r$일 때 성분 $<X, p, q>$가 주어지지 않은 상태에서 성분 $<Y, q, r>$이 먼저 주어졌다고 가정해 보자. 문법 규칙 $C \rightarrow Y\gamma$로부터 새로운 활성 아크 $<C \rightarrow Y \cdot \gamma, q, r>$을 생성하는 데에는 아무런 문제가 없다. 하지만 성분 $<X, p, q>$가 주어지지 않아 아직은 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$가 생성되기 전이다. 따라서 이 상황에서 성분 $<Y, q, r>$이 주어진다 하더라도 활성 아크 $<A \rightarrow \alpha XY \cdot \beta, o, r>$이 생성될 수는 없다.

이제, 성분 $<Y, q, r>$에 의한 활성 아크의 확장이 완료된 후에 성분 $<X, p, q>$가 주어졌다고 해보자. 그러면 활성 아크 $<A \rightarrow \alpha \cdot XY\beta, o, p>$로부터 새로운 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$가 생성될 것이다. 그런데 성분 $<Y, q, r>$이 성분 $<X, p, q>$에 앞서 이미 주어졌으며, 성분 $<Y, q, r>$에 의한 활성 아크의 확장은 이미 끝난 상황이기 때문에 성분 $<X, p, q>$에 의해 생성된 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$를 $<A \rightarrow \alpha XY \cdot \beta, o, r>$로 확장할 수는 없다. 이 상황을 그림으로 나타내면 그림 3과 같다.

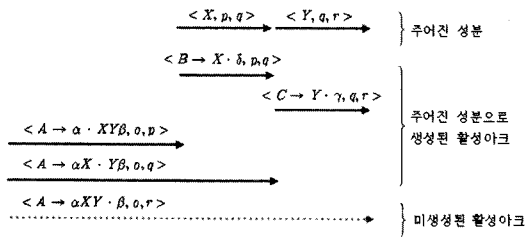


그림 3 비순차적 파싱으로 인해 활성 아크 생성에 실패하는 경우

그림 3에서 파선으로 표시한 것은 입력 문장의 왼쪽에서 오른쪽으로 순차적인 파싱을 할 때에는 생성되지만 비순차적인 파싱을 하면서 생성에 실패한 활성 아크를 나타낸다.

위에서는 설명의 편의를 위하여 인접한 두 성분 $<X, p, q>$와 $<Y, q, r>$을 가지고 비순차적으로 파싱이 진행될 때 발생하는 문제점에 대하여 설명하였다. 하지만 이러한 문제점은 인접한 두 성분 사이에서만 발생하는 것은 아니다. 일반적으로 말해서 성분 $<Y, r, s>$를 가지고 활성 아크를 확장하는 작업이 모두 끝난 후 이보다 왼쪽에 오는 다른 성분 $<X, p, q>$(단, $q<r$)를 가지고 활성 아크를 확장하는 작업을 할 경우, 순차적인 파싱에서는 잘 생성되던 활성 아크가 생성되지 않아 정상적인 파싱을 할 수 없다는 문제가 발생한다. 이러한 문제가 있기 때문에 그림 1과 같이 주어진 차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 순차적인 파싱을 해야 한다는 제약이 따르는 것이다.

3.3 임의 순서 차트 파싱 알고리즘

위에서 설명한 바와 같이 차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행해야 한다는 제약이 따른다. 이러한 제약 조건을 제거하는 방안을 찾기 위해 그림 1의 차트 파싱 알고리즘에서 어젠더에 저장된 성분을 하나씩 꺼내 활성 아크를 확장한 후 차트에 저장한다는 점에 주목을 해보자. 차트에 저장된 성분들은 파싱 과정에서 활성 아크를 확장하는 데 이미 사용된 것들이다. 그러므로 순차적으로 파싱을 진행할 때에는 잘 생성되던 활성 아크가 임의 순서로 파싱을 할 경우에는 생성되지 않는 문제는 활성 아크가 생성될 때마다 차트에 저장된 성분으로 추가 확장의 가능성을 조사하고 가능한 경우 추가 확장하는 방법으로 해결할 수 있을 것이다.

성분 $<Y, q, r>$에 의한 활성 아크의 확장이 끝나면 성분 $<Y, q, r>$은 차트에 저장된다. 이후에 이보다 왼쪽에 오는 성분 $<X, p, q>$가 주어진 경우를 생각해 보자. 성분 $<X, p, q>$에 의해 활성 아크 $<A \rightarrow \alpha \cdot XY\beta, o, p>$는 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$로 확장된다. 기존 차트 파싱 알고리즘에서는 성분 $<Y, q, r>$에 의한 확장이 이미 끝난 상황이기 때문에 이 활성 아크를 $<A \rightarrow \alpha XY \cdot \beta, o, r>$로 확장하는 것이 불가능하였다. 하지만 우리는 위에서 언급한 바와 같이 활성 아크가 생성될 때마다 차트에 저장된 성분을 이용한 추가 확장의 가능성을 조사하고, 추가 확장이 가능한 경우 새로운 활성 아크를 생성하려고 한다.

활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$를 추가 확장할 수 있는 성분은 q에서 시작하는 성분 Y로 제한된다. 그런데 이러한 조건에 맞는 성분 $<Y, q, r>$이 이미 차트에 저장되어 있으므로 이 성분을 이용하여 활성 아크 $<A \rightarrow \alpha X \cdot Y\beta, o, q>$를 $<A \rightarrow \alpha XY \cdot \beta, o, r>$로 추가 확장할 수 있다. 이와 같이 하면 왼쪽에서 오른쪽으로 파싱이 진행되어야 한다는 제약 조건을 지키지 않았을 때 기존 차트 파싱 알고리즘으로는 생성할 수 없었던 활성 아크를 생성할 수 있게 된다. 그림 4는 성분 $<Y, q, r>$이 먼저 주어지고 성분 $<X, p, q>$가 나중에 주어진 경우 차트 성분으로 활성 아크를 추가 확장한

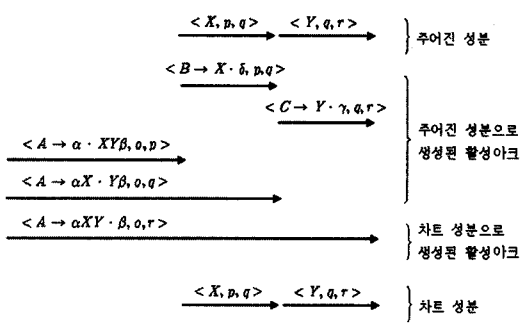


그림 4 차트에 저장된 성분으로 활성 아크를 추가 확장한 후 상황을 보여주고 있다. 이것을 그림 2와 비교하면 순차적으로 파싱을 할 때와 동일한 개수의 활성 아크가 생성됨을 알 수 있다.

차트에 저장된 성분을 이용하여 활성 아크를 추가 확장하는 절차는 한 번으로 끝나는 것이 아니라 더 이상 추가 확장을 할 수 없을 때까지 동일한 방법으로 확장을 해 나간다. 그래서 차트에 저장된 성분 $< Y, q, r >$을 이용하여 활성 아크 $< A \rightarrow \alpha X \cdot Y\beta, o, q >$를 $< A \rightarrow \alpha XY \beta, o, r >$로 추가 확장한 후 r에서 시작하고 β와 매치되는 또 다른 성분이 차트에 존재한다면 마찬가지 방법으로 활성 아크를 계속 확장해 나간다.

```
1 randomOrderChartParsing(sentence of n words)
2 {
3     for i from 0 to n−1 do
4         add constituents < X, i, i+1 > to agenda in any order you choose
5     end
6
7     for any grammar rule of the form S→α do // assume S is a starting symbol
8         add an initial active arc < S→ · α, 0, 0 > to chart
9     end
10
11    while agenda is not empty do
12        given a constituent < X, p, q > ← pop(agenda) do
13            for all active arc of the form < A →α · X β, o, p > in chart do
14                if β is empty then
15                  add a new constituent < A, o, p > to agenda
16                else
17                  add a new active arc < A →αX · β, o, q > to chart
18                  extendActiveArcWithChart(< A →αX · β, o, q >)
19                end
20            end
21            for a grammar rule B→Xδ do
22                if δ is empty then
23                  add a new constituent < B, p, q > to agenda
24                else
25                  add a new active arc < B→X · δ, p, q > to chart
26                  extendActiveArcWithChart(< B→X · δ, p, q >)
27                end
28            end
29            add the constituent < X, p, q > to chart
30        end
31    end
32 }
33
34 extendActiveArcWithChart(active arc < A →α · Yβ, i, j >)
35 {
36      for all constituents < Y, j, k > in chart do
37          create a new active arc of the form < A →αY · β, i, k >
38          if β is empty then
39              add a new constituent < A, i, k > to agenda
40          else
41              add the active arc < A →αY · β, i, k > to chart
42              extendActiveArcWithChart(< A →αY · β, i, k >)
43          end
44      end
45 }
```

그림 5 임의 순서 차트 파싱 알고리즘

지금까지 설명한 방법으로 파싱을 한다면 입력 문장의 왼쪽에서 오른쪽으로 순차적으로 파싱을 하지 않을 경우 기존 차트 파싱 알고리즘으로는 생성되지 않았던 활성 아크들을 모두 생성할 수가 있다. 그림 5는 이 방법을 형식화하여 나타낸 임의 순서 차트 파싱 알고리즘이다. 이 알고리즘은 그림 1의 차트 파싱 알고리즘을 수정한 것이다.

3.4 임의 순서 차트 파싱 알고리즘의 성능

임의 순서 차트 파싱 알고리즘의 성능에 대하여 논하기 위하여 기존 차트 파싱 알고리즘과 임의 순서 차트 파싱 알고리즘에서의 활성 아크 확장 과정을 비교해 보자. 기존 차트 파싱 알고리즘에서는 새로운 성분 X가 주어지면 그 때까지 생성된 활성 아크 중에서 확장 가능한 것들을 모두 찾아 확장하는 방식으로 진행되는데, 그림 6과 같이 활성 아크들을 구간별, 종류별로 분류하여 저장해 둔다면 성분 X를 가지고 활성 아크를 확장하는 작업을 효율적으로 진행할 수 있다. 이 경우 활성 아크 확장 작업은 성분 X가 주어지기를 기다리고 있는 활성 아크의 개수만큼 반복되는데, 이것을 그림 6에서 가는 파선으로 표시하였다.

임의 순서 차트 파싱 알고리즘에서 인자로 활성 아크 $A\rightarrow\alpha\cdot Y\beta$가 주어졌을 때 extendActiveArcWithChart() 함수에서 활성 아크가 추가 확장되는 과정을 살펴보자. 그림 5의 36행에서 보듯이 이 함수에서는 차트에 저장된 성분 Y 중에서 주어진 활성 아크를 확장할 수 있는 구간에서 시작하는 성분들을 대상으로 활성 아크를 확장한다. 활성 아크를 효율적으로 확장하기 위하여 파싱 과정에서 생성된 성분을 차트에 저장할 때에는 그림 7과 같이 특정 위치에서 시작하는 성분을 종류별로 분류하여 저장해 둔다. 이와 같이 차트를 관리하면 주어진 활성 아크 $A\rightarrow\alpha\cdot Y\beta$를 확장하는 작업은 차트에 저장된 성분 Y의 개수만큼 반복되는데, 이것을 그림 7에서 가는 파선으로 표시하였다.

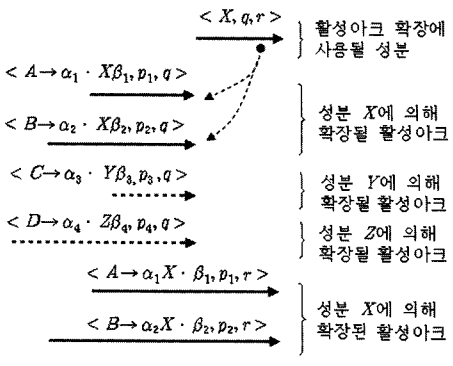


그림 6 차트 파싱 알고리즘에서의 활성 아크 확장

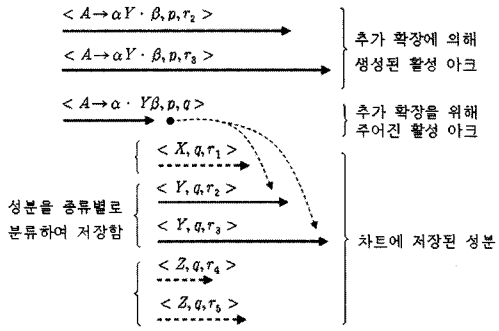


그림 7 extendActiveArcWithChart() 함수에서의 활성 아크 추가 확장

차트 파싱에서는 주어진 성분 X로 확장 가능한 모든 활성 아크를 확장하는 데 반하여, 임의 순서 차트 파싱에서 주어진 활성 아크에 대하여 이것을 확장할 수 있는 모든 차트 성분을 찾아 주어진 활성 아크를 확장한다. 이는 무엇을 기준으로 새로운 활성 아크 생성 작업을 반복하는가에서 주체만 바뀌었을 뿐 활성 아크를 확장하는 과정을 반복하는 회수는 동일하다. 따라서 동일한 문법, 동일한 문장에 대하여 차트 파싱 알고리즘으로 분석하든 임의 순서 차트 파싱 알고리즘으로 분석하든 생성되는 성분과 활성 아크의 총 갯수는 같아야 한다. 실제로 625개의 문법 규칙으로 이루어진 중국어 문법으로 285 문장을 임의 순서 차트 파싱 알고리즘으로 분석했을 때 기존 차트 파싱 알고리즘으로 분석할 때와 마찬가지로 80,269개의 성분과 1,824,994개의 활성 아크가 생성되었다.

그런데 임의 순서 차트 파싱 알고리즘에서는 활성 아크가 생성될 때마다 extendActiveArcWithChart() 함수를 호출하여 추가 확장을 하게 되는데, 파싱 과정에서 생성되는 활성 아크의 수가 적지 않으므로 함수 호출로 인한 부담(overhead)이 발생하게 된다. 또 성분을 차트에 저장할 때에는 그림 7과 같이 각 성분을 종류별로 분류하여 저장해야 하는 데에 따른 추가적인 부담도 발생한다. 이러한 이유 때문에 임의 순서 차트 파싱의 수행 속도는 기존 차트 파싱보다 약간 느릴 수밖에 없다. 위에서 사용한 것과 동일한 문법으로 285 문장을 분석하고 그 시간을 측정하는 실험을 해 본 결과, 임의 순서 차트 파싱은 기존 차트 파싱에 비하여 약 8% 가량 수행 시간이 더 걸린 것으로 나타났다.

4. 임의 순서 차트 파싱 알고리즘의 적용 예

여기에서는 다음과 같은 간단한 문법을 가지고 앞에서 설명한 임의 순서 차트 파싱 알고리즘을 적용하는 예를 보기로 한다.

S → NP VP	VP → AUX VP	NP → ART N
S → AUX S	VP → V NP	

이 문법으로 입력 문장 "a can can can a can"의 왼쪽에서 오른쪽으로 기존 차트 파싱 알고리즘에 따라 파싱을 하면 그림 8과 같이 활성 아크 및 성분이 생성된다. 여기서는 편의상 입력 문장의 각 단어가 하나의 품사만 가지는 것으로 가정하였다.

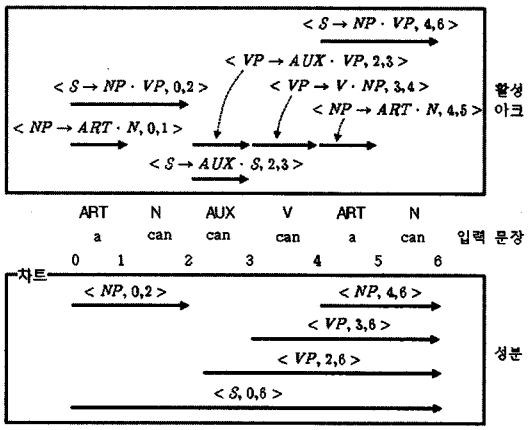


그림 8 차트 파싱 알고리즘에 의한 좌우 파싱

다음은 차트 파싱 알고리즘을 사용하되 입력 문장의 왼쪽에서 오른쪽으로 파싱을 하지 않고 임의 순서로 파싱을 할 경우 어떻게 되는지 살펴보자. 3.2절에서 설명한 바에 따르면 이런 경우에는 정상적으로 활성 아크와 성분이 생성되지 않아 파싱에 실패하게 된다. 여기에서는 설명의 편의를 위하여 다음과 같은 순서로 파싱을 진행하는 것으로 가정한다.

a	can	can	can	a	can
①	②	⑥	⑤	③	④

차트 파싱 알고리즘을 사용하여 위와 같은 순서로 파싱을 할 때 생성되는 활성 아크와 성분은 그림 9와 같다.

이 그림에서 활성 아크 $< VP \rightarrow AUX \cdot VP, 2,3 >$과 $< VP \rightarrow V \cdot NP, 3,4 >$는 성분 $< NP, 4,6 >$이 생성된 이후에 만들어지기 때문에 더 이상의 확장이 불가능하며, 이로 인해 $< VP, 3,6 >$이나 $< VP, 2,6 >$ 등과 같은 성분이 생성되지 않는다. 이처럼 기존 차트 파싱 알고리즘으로는 입력 문장의 왼쪽에서 오른쪽으로 파싱을 진행하지 않을 경우 정상적으로 파싱이 되지 않는다.

임의 순서 차트 파싱 알고리즘에서는 활성 아크가 생성될 때마다 차트에 저장된 성분으로 추가 확장을 한다

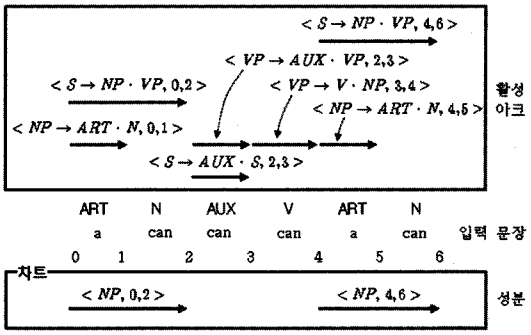


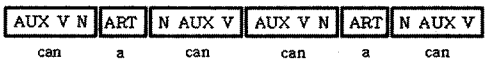
그림 9 차트 파싱 알고리즘에 의한 임의 순서 파싱

고 했다. 그러므로 가령 활성 아크 $< VP \rightarrow V \cdot NP, 3,4 >$가 생성되었을 때 차트에 $< NP, 4, * >$와 같은 형태의 성분이 있다면 이 활성 아크를 추가 확장할 수 있다.[1] 그런데 위의 예에서는 이러한 조건에 맞는 성분 $< NP, 4,6 >$이 차트에 저장되어 있으므로 활성 아크 $< VP \rightarrow V \cdot NP, 3,4 >$가 추가 확장되며, 그 결과 새로운 성분 $< VP, 3,6 >$이 생성된다. $< VP, 3,6 >$가 생성되면 $< VP \rightarrow AUX \cdot VP, 2,3 >$이 확장되어 $< VP, 2,6 >$이 생성되며 다시 $< S \rightarrow NP \cdot VP, 0,2 >$가 확장되어 $< S, 0,6 >$이 생성되므로 파싱이 성공적으로 마무리된다.

이번에는 입력 문장 "can a can can a can"에 대한 형태소 분석 및 태깅 결과가 다음과 같이 주어졌다고 해보자. 여기서 각 단어별 형태소 분석 결과는 태깅 점수가 가장 큰 것부터 정렬되어 있다고 가정한다.

can	a	can	can	a	can
AUX	ART	N	AUX	ART	N
V		AUX	V		AUX
N		V	N		V

기존 차트 파싱 알고리즘에서는 왼쪽에서 오른쪽으로 파싱을 진행해야 하므로 형태소 분석 결과를 아래 그림과 같이 순서대로 어젠더에 저장한 후 파싱을 시작한다. 따라서 한 단어에 대한 모든 형태소 분석 결과를 가지고 문법 규칙을 적용하고 활성 아크를 생성하는 등의 작업을 한 후에 다음 단어로 넘어가는 것이다. 이 과정에서 많은 메모리를 필요로 하기 때문에 긴 문장이 주어진 경우에는 왼쪽에서 오른쪽으로 파싱을 진행하는 도중에 메모리가 다 소진되어 문장 중간에서 파싱을 중단해야 하는 상황이 발생할 수도 있다.



1) $< NP, 4, * >$는 입력 구간 4에서 시작하는 임의의 NP를 나타낸다.

이러한 문제 때문에 [2]에서는 각 단어별로 태깅 점수가 가장 높은 형태소만 가지고 기존 차트 파싱 알고리즘으로 파싱을 하였다. 품사 태거의 정확도가 100%라고 한다면 이런 방법으로 파싱을 해도 별 문제가 없다. 하지만 현실적으로 품사 태거의 정확도가 97% 내외이므로 문장의 길이가 길면 길수록 태깅 오류가 발생할 가능성이 높아지고 결과적으로 파싱에 실패할 가능성도 증가하게 된다. 위의 예에서 태깅 점수가 가장 높은 품사 중에는 잘못 된 것도 있는데, 이것을 차트 파싱 알고리즘에 따라 파싱을 하면 그림 10에서 보듯이 파싱에 실패하게 된다.

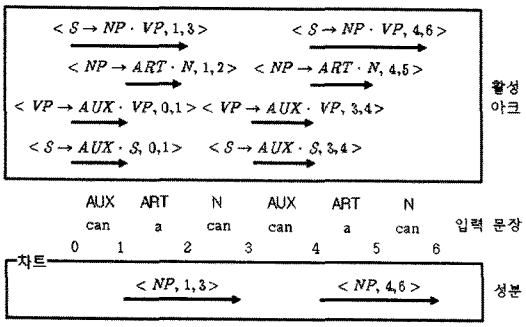


그림 10 최적 품사만으로 파싱했을 때 실패한 예

이 상황에서 차순위 품사를 가지고 파싱을 계속하려 해도 기존 차트 파싱 알고리즘에서는 입력 문장의 왼쪽에서 오른쪽으로 순차적으로 파싱을 해야 한다는 제약이 있으므로 더 이상 파싱을 진행할 수가 없다. 하지만 임의 순서 차트 파싱 알고리즘에서는 형태소 분석 결과를 임의 순서로 어젠더에 저장하고 파싱을 할 수 있으므로 일단 태깅 점수가 가장 높은 품사열을 가지고 파싱을 수행하되, 만약 파싱에 실패할 경우에는 차순위 품사를 점진적으로 추가하면서 파싱을 계속할 수 있다.

그림 10에서 네 번째 단어가 AUX로 잘못 태깅되었기 때문에 파싱에 실패하였다. 그래서 이번에는 차순위 품사인 V를 가지고 파싱을 계속 진행해 보기로 한다. 네 번째 단어의 차순위 품사로 추가된 V에 의해 활성 아크 $< VP \rightarrow V \cdot NP, 3, 4 >$가 생성되며, 이 활성 아크는 차트에 저장된 성분 $< NP, 4, 6 >$에 의해 추가 확장되어 새로운 성분 $< VP, 3, 6 >$이 만들어진다. 이 성분은 다시 활성 아크 $< S \rightarrow NP \cdot VP, 1, 3 >$을 확장하여 $< S, 1, 6 >$이 만들어진다. 이와 같이 임의 순서 차트 파싱 알고리즘을 사용하게 되면 일단 최적 품사열로 파싱을 하되 파싱에 실패할 경우에는 차순위 품사를 점진적으로 추가해 가면서 파싱을 할 수 있기 때문에 긴 문장이 주어진 경우에도 파싱 도중에 메모리 부족으로 중단해야 하는 상황이 발생할 가능성은 상대적으로 낮아진다. 그림 11은 네 번째 단어의 차순위 품사 V를 추가하여 임의 순서 차트 파싱 알고리즘으로 파싱을 계속한 결과이다.

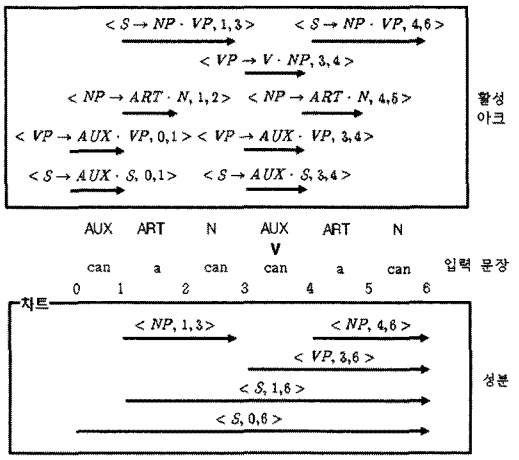


그림 11 차순위 품사로 임의 순서 차트 파싱을 한 결과

앞에서 사용한 285 문장으로 본 논문에서 제안한 파싱 알고리즘을 사용할 경우 실질적으로 어떤 장점이 있는지를 평가하는 실험을 수행하였다. 각 단어별로 태깅 점수가 가장 높은 형태소 하나씩을 선택하여 파싱했을 때에는 전체 285 문장 중에서 33 문장은 파싱에 실패하였다.[2] 이는 품사 태거 정확도가 100%가 아니기 때문에 나타나는 문제점이다. 실제 응용에서는 파싱 실패가 대단히 치명적이기 때문에 한 단어에 대하여 복수의 형태소를 가지고 파싱을 하는 방법을 취한다. 하지만 이때에는 필요 이상으로 많은 파스 트리가 생성된다는 문제점이 나타난다. 한편, 본 논문에서 제안한 임의 순서 차트 파싱 알고리즘을 사용하면 일단은 가장 태깅 점수가 높은 형태소 하나만으로 파싱을 하되 만약 파싱에 실패하게 되면 차순위 형태소를 하나씩 하나씩 점진적으로 추가하는 방식으로 파싱을 할 수 있으므로 최소의 파스 트리를 생성하면서도 파싱 실패가 발생하지 않도록 할 수 있다. 표 1은 285 문장에 대하여 이 세 가지 방법으로 파싱을 했을 때 생성되는 활성 아크, 성분, 파스 트리의 평균 개수를 비교한 것이다.

이 표에서 보듯이 점진적으로 파싱을 할 경우에는 파싱에 실패한 문장이 하나도 없으면서도 파싱 과정에서 생성된 활성 아크, 성분, 파스 트리의 개수는 단일 형태

2) 285 문장의 평균 길이는 13.5 단어인데 반하여 파싱에 실패한 33 문장의 평균 길이는 16.8 단어로 문장이 길어질수록 파싱에 실패할 가능성도 높아짐을 알 수 있다.

표 1 파싱 방식에 따른 성능 비교

파싱방식 / 비교항목	복수 형태소 파싱	단일 형태소 파싱	점진적 파싱
활성 아크	6404 개/문장	3084 개/문장	3365 개/문장
성분	282 개/문장	116 개/문장	129 개/문장
파스 트리	43.5 개/문장	11.2 개/문장	13.0 개/문장
파싱 실패	0 문장	33 문장	0문장

소로 파싱을 할 때보다 약간 증가하는 정도이다. 단일 형태소로 파싱을 할 때 발생하는 파싱 실패 문제를 피하기 위하여 복수 형태소로 파싱을 한다면 점진적 파싱에 비하여 2-3배나 많은 활성 아크, 성분, 파스 트리가 생성되므로 비효율적이다.

5. 결 론

긴 문장에 대한 대처 방안으로 태깅 점수가 가장 높은 후보만으로 파싱을 하는 방법이 일반적으로 많이 사용되고 있다. 그런데 태거의 정확도가 100%에는 못 미치기 때문에 문장 길이가 긴 경우에는 태깅 오류가 포함될 가능성이 높다. 태깅 오류로 인해 파싱에 실패할 경우 왼쪽에서 오른쪽으로 파싱이 진행되어야 한다는 제약 조건을 가지고 있는 기존 차트 파싱 알고리즘으로는 태깅 결과의 차순위 후보를 가지고 파싱을 계속하는 것이 불가능하다. 본 논문에서는 이러한 제약 조건을 없앤 임의 순서 차트 파싱 알고리즘을 제안하였다. 제안한 알고리즘에서는 입력 문장의 각 단어에 대하여 어떤 순서로 파싱을 진행하더라도 파싱이 가능하므로 태거의 최적 결과만으로 파싱에 실패할 경우 차순위 후보를 점진적으로 추가해 가면서 파싱을 계속하는 것이 가능하다.

참 고 문 헌

[1] Martin Kay, "Algorithm schemata and data structures in syntactic processing," Technical Report CSL80-12, Xerox PARC, Palo Alto, 1980.

[2] 구조설계 : 영한 기술논문 자동번역 시스템 (v1.0), 음성/언어정보연구센터, 한국전자통신연구원, 2007.

[3] Oliviero Stock, Rino Falcone and Patrizia Insinnamo, "Island Parsing and Bidirectional Chart," *Proceedings of the 12th Conference on Computational Linguistics*, vol.2, pp.636-641, 1988.

[4] Aristomenis Thanopoulos, Nikos Fakotakis and George Kokkinakis, "Linguistic Processor for a Spoken Dialogue System based on Island Parsing Techniques," *Proceedings of 5th Eurospeech*, vol.4, pp.2259-2262, 1997.

[5] Giorgio Satta and Oliviero Stock, "Bidirectional Context-free Grammar Parsing for Natural Language Processing," *Artificial Intelligence*, vol.69, pp.123-164, 1994.

[6] Bernd Kiefer, "Redundancy-free Island Parsing of Word Graphs," *19th International Joint Conference on Artificial Intelligence*, pp.1079-1084, 2005.

[7] Sung-Wan Park and Dong-Yul Ra, "A Robust Natural Language Parsing Method Using Local Bi-directional Analysis," *Proceedings of the 27th KIISE Fall Conference*, vol.27, no.2, pp.176-178, 2000. (in Korean)

[8] Jong C. Park, Hyun Sook Kim and Jung Jae Kim, "Bidirectional Incremental Parsing for Automatic Pathway Identification with Combinatory Categorial Grammar," *Proceedings of Pacific Symposium on Biocomputing*, pp.396-407, 2001.

[9] James Allen, Natural Language Understanding, 2nd edition, The Benjamin/Cummings Publishing Company, Inc., 1995.

[10] Daniel Jurafsky and James Martin, Speech and Language Processing, Prentice Hall, 2000.

심 광 섭

1986년 서울대학교 컴퓨터공학과 졸업 1988년 서울대학교 컴퓨터공학과 석사학위 취득. 1994년 서울대학교 컴퓨터공학과 박사학위 취득. 1995년~현재 성신여자대학교 IT학부 재직. 관심분야는 자연어처리, 기계번역, 정보검색, 인공지능 등