Journal of the Korea Industrial Information Systems Research Vol. 24 No. 2, Apr. 2019 : 1-14
http://dx.doi.org./10.9723/jksiis.2019.24.2.001 ISSN:1229-3741

트리구조의 문서에 대한 편집스크립트 조정+

(Adjusting Edit Scripts on Tree-structured Documents)

이 석 균[1)*], 엄 현 민[2)]
(Lee SukKyoon and Um HyunMin)

요 약 웹, XML, 오피스 어플리케이션에 사용되는 대부분의 문서들은 트리 구조로 구성되어 있으며 특히 다중 사용자 환경에서의 트리 구조의 문서의 차이 발견, 합병, 버전 제어 등의 연구가 활발하다. 그러나 이들의 기초가 되는 편집스크립트에 대한 연구는 초보적인 상태에 있다. 본 논문에서는 편집연산들의 실행 시 트리구조의 문서의 변화를 이해하기 위한 문서 모델을 제시하고 편집연산들의 실행 효과의 분석을 통해 트리 구조 문서에 대한 인접한 편집연산들의 순서 교환 방법을 제안한다.

트리 구조 문서에 대한 변화탐지의 결과로 생성되는 편집스크립트들은 대부분 기본연산들(갱신, 삽입, 삭제)만으로 구성된다. 그러나 이동, 복사연산을 포함하는 경우, 이들의 복합연산의 특성으로 인해 주로 2단계 패스의 실행을 전제로 하는 편집스크립트를 생성한다. 본 논문에서는 제안한 편집연산들의 순서 교환 방법을 통해 2단계 패스의 실행을 전제로 하는 X-treeESgen의 편집스크립트를 단일 패스로 변환하는 알고리즘을 제시한다.

핵심주제어 : 편집스크립트, Diff, 트리구조 문서, XML, 버전관리

Abstract Since most documents used in web, XML, office applications are tree-structured, diff, merge, and version control for tree-structured documents in multi-user environments are crucial tasks. However research on edit scripts which is a basis for them is in primitive stage. In this paper, we present a document model for understanding the change of tree-structured documents as edit scripts are executed, and propose a method of switching adjacent edit operations on tree-structured documents based on the analysis of the effects of edit operations.

Mostly, edit scripts which are produced as the results of diff on tree-structured documents only consist of basic operations such as update, insert, delete. However, when move and copy are included in edit scripts, because of the characteristics of their complex operation, it is often that edit scripts are generated to execute in two passes. In this paper, using the proposed method of switching edit operations, we present an algorithm of transforming the edit scripts of

* Corresponding Author: sklee@dankook.ac.kr
Manuscript received December 23, 2018 / revised January 22, 2019 / accepted January 31, 2019
+ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 SW 중심대학지원사업의 연구결과로 수행되었음(2017-0-00091).
1) 단국대학교 SW융합대학 소프트웨어학과, 제 1저자
2) 단국대학교 공과대학 소프트웨어학과, 제 2저자

X-treeESgen, which are designed to execute in two passes, into the ones that can be executed in one pass.

Key Words : Edit Script, Diff, Tree-structured documents, XML, Version management

1. 서 론

현대 사회에서는 정보는 문서들을 통해 저장, 전달 및 교환되며 문서들은 애플리케이션들에 따라 다양한 형태로 생성되고 관리된다. 모바일 기기의 확산과 네트워크의 발전으로 문서들의 생성과 관리는 종종 분산 환경에서 협업 형태로 이루어진다. 이러한 작업 환경에서 문서들의 편집이 다수의 참여자들에 의해 동시에 이루어질 경우, 내용의 일관성을 유지하면서 시간 흐름에 따른 문서들의 버전들을 추적할 수 있는 버전 관리 기술은 중요한 이슈로 등장한다[1,2].

텍스트 문서들의 경우 diff, patch 등이 버전 관리의 핵심 기술로 사용되어왔으나, 텍스트 문서들은 라인 단위를 전제로 문자 또는 문자열의 비교를 전제로 하고 있어 이들을 트리 구조를 전제로 하는 최근의 문서들에 적용하는 것은 적절하지 않다[3]. 웹 문서들은 HTML로 표현되고 대부분의 오피스 문서들은 XML 기반의 소스 파일의 형식을 지원하고 있으며, 대부분의 애플리케이션들에서 데이터들은 종종 XML 또는 JSON으로 저장되는데, HTML, XML, 그리고 JSON으로 표현되는 문서들은 모두 트리 구조로 표현된다[4,5]. 최근 트리 구조의 문서들에 대한 diff, patch, 버전 제어 등에 대한 연구가 활발히 진행되고 있다[3]. 이 중 가장 기초 기술은 diff로 문서의 '차이' 즉 변화 내용(delta)을 추출하는 작업이고 이때 문서의 차이는 일련의 편집연산들, 즉 편집스크립트로 표현되곤 한다.

트리구조 문서의 변화 탐지에 대해 Selkow[6]는 의해 트리-대-트리 편집(tree-to-tree editing) 문제로 정의하고 변화 내용을 삽입, 삭제, 갱신연산들로 구성된 편집스크립트로 표현하였고, Zhang과 Shasha[7]는 순서 트리들에 전제로 편집거리(editing distance)개념을 사용하여 개선된 알고리즘을 제안하였다. 90년대 이후에는 실시간 처리를 위한 휴리스틱 기반의 알고리즘들[4,8-14]가 개발되었는데, 대표적인 알고리즘들로는 XyDiff[8], X-treeDiff+[9-10], XMDiff[11], DiffXML[12], DeltaXML[13-14], XCC diff[4] 등이 있다. 이들 중 XMDiff, DeltaXML은 기본(편집)연산들 즉 삽입, 삭제, 갱신연산들만을, XyDiff, DiffXML와 XCC diff는 기본연산들 외에 이동연산을, 그리고 X-treeDiff+는 기본연산들 외에 이동연산과 복사연산을 지원한다. 최근에는 트리구조에 대한 일반적인 diff보다는, 시맨틱 기반의 diff, 또는 애플리케이션 분야에 특화된 diff 연구가 이루어졌다. Dohrn[15]는 오피스 문서와 같이 텍스트의 변화가 중요한 경우 텍스트 노드의 변화에 초점을 둔 HDDiff를, Autexier[16]은 시맨틱 기반의 유사성(similarity)를 고려한 SDIFF을, 그리고 Antila외 2인[2]은 음악 분야의 특성을 고려해 협업 기반의 악보 제작을 위한 diff 알고리즘을 제안했다.

사용자들은 문서 편집과정에서 기본연산들과 이동연산, 복사연산을 사용하고 편집스크립트의 편집연산들은 순차적 실행을 전제로 하며 항상 직전 연산의 실행결과의 트리에 대한 노드 경로를 사용한다. 위에서 언급한 대부분의 diff 연구 결과들은 편집스크립트 생성 방법을 자세히 설명하고 있지 않으며, 기본연산들 외에 이동연산을, 또는 이동연산과 복사연산을 지원하는 알고리즘들의 편집스크립트들은 애플리케이션을 통해 실행하도록 구성된다. XCC diff는 집합기반의 델타(set-based delta) 개념의 편집스크립트를 생성하는데, 편집연산들의 대상 노드들의 경로가 직전 연산의 실행 결과의 트리가 아니라 원본 트리의 노드들을 참조한다[4]. XyDiff의 구현소스를 분석하면 X-treeDiff+의 편집스크립트의 생성과 유사한 2단계 패스의 실행을 전제로 한다. X-treeDiff+의 편집스크립트는 X-treeESgen을 통해 생성하

는데, 이는 기본연산들은 순차적 처리를 전제로 생성하지만 복합연산인 이동연산(삽입과 삭제)과 복사연산(복사와 붙여넣기)을 2단계 패스의 실행을 전제로 생성한다[17]. 따라서 위의 두 방법 모두 순차적 처리에 익숙한 일반 사용자가 이해하기는 쉽지 않다.

본 논문에서는 편집연산들의 실행 시 트리구조의 문서의 변화를 이해하기 위한 문서 모델을 제안하고 이 변화가 후속 연산에 끼치는 영향을 분석한다. 이를 기반으로 편집스크립트 내에서의 인접한 편집연산들의 순서교환에 필요한 조건과 조정 방법을 제안하고 적용 사례로 X-treeESgen의 편집스크립트를 순차적 실행이 가능한 편집스크립트로 변환한 결과를 소개한다. X-treeDiff+ 기반의 유사한 연구[18]이 시도되었으나 이론적인 근거를 충분히 제시하지 못하고 논리 흐름에 한계가 있었다.

본 논문에서 제안된 편집연산들의 조정 방법은 확장을 통해 다른 diff 알고리즘들에 의해 생성된 편집스크립트들을 일반 사용자가 이해할 수 있는 순차적 흐름의 편집스크립트로의 변환 작업, 그리고 편집스크립트 내의 인접한 편집연산들의 교환 및 병합 등을 통해 간결한 편집스크립트로의 변환 작업 등이 가능하다. 이는 협업을 통한 문서 작성 시 생성된 편집스크립트들의 합병 등의 연구에 기초가 된다. 본 논문의 구성은 다음과 같다. 2절은 본 논문에서 사용할 문서 모델과 편집연산의 유형들, 그리고 X-treeESgen의 특징과 문제점을 제시하고, 3절에서는 인접한 기본연산들에 대한 순서교환의 조건과 조정방법을, 4절에서는 X-treeESgen의 편집스크립트를 비단조적 증감 구조의 단일 패스 편집스크립트로 변경하는 알고리즘을 제안하고 이에 대한 분석 및 특징을 설명하고 5절에서 결론을 제시한다.

2. 연구 배경

본 절에서는 문서 모델과 편집연산의 유형들을 소개하고 X-treeESgen 알고리즘의 개요, 특징과 문제점을 설명한다.

2.1 문서 모델과 편집연산 유형

본 논문에서는 트리 구조의 문서의 표현을 위해 Fig. 1과 같은 레이블 트리 형태의 문서 모델을 사용한다.

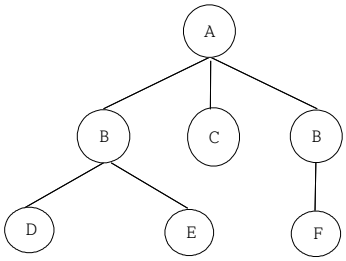

Fig. 1 Example of Source Tree

트리에서의 노드 식별을 위해 XML의 XPath가 사용될 수 있는데, XPath는 노드의 레이블과 동일 레이블의 노드들 사이의 형제 순서를 통해 노드를 식별한다. 그러나 XPath는 동일 레이블의 노드 간의 형제 순서의 식별에는 편리하나 다른 레이블 노드들과의 형제 순서는 표현할 수 없다는 단점이 있다. 가령 Fig.1에서 XPath로 표현된 두 노드 즉 /A[1]/B[1]과 /A[1]/C[1]들은 서로의 상대적 위치를 알 수 없다. 본 논문에서는 이를 개선한 노드의 식별 방법으로 tPath를 사용한다. tPath는 노드의 레이블과 형제 노드들의 순서 정보가 노드의 경로 표현에 사용된다. Fig.1의 /A[1]/B[1]과 /A[1]/C[1]를 tPath를 표현하면 이들은 /A(1)/B(1)와 /A(1)/C(2)로, 그리고 레이블이 F인 노드는 /A(1)/B(3)/F(1)으로 표현된다.

Table 1은 본 논문의 설명에 사용될 편집연산들 U, D, I, C, M을 제시하는데, 이들은 레이블 트리를 전제로 X-treeESgen의 편집연산들을 단순화하여 표현했다. X-treeESgen은 원본트리와 결과트리에 X-treeDiff+을 적용 후 생성된 대응정보를 기반으로 편집스크립트를 생성하고, 원본트리에 편집스크립트를 적용하면 결과트리가 생성된다. 텍스트 노드의 수정과 속성 관련 편집연산들을 트리의 구조를 변화시키지 않으므로 구별없이 U로 표시하고 모든 편집연산들에는 각 편집연산을 식별할 수 있는 ID 속성을 포함되어 있다고 가정한다. 편집연산들 중 U, D, I는 기본

Table 1 List of Edit Operations

Operation	Explanation
U(n, v)	Update operation: update the value of node n by v.
D(n)	Delete operation: delete the node n.
I(c, n, p, i_ord)	Insert operation: increate a new node with contents of c, and insert the new node into node p as p^{th} child of p. i_ord represent the insertion order.
C(sn, tn, tp, i_ord)	Copy operation: copy the subtree rooted at sn, and paste the subtree as tp^{th} child of tn. i_ord represent the insertion(pasting) order.
M(sn, tn, tp, i_ord)	Move operation: delete the subtree rooted at sn, and paste the subtree as tp^{th} child of tn. i_ord represent the insertion(pasting) order.

* Note : n, sn and tn represents a node. p is an insertion position of n and tp a pasting position of tn.

연산이라고 하고 C는 복사(copy)와 붙여넣기(paste), 그리고 M은 잘라내기(cut)와 붙여넣기(paste)의 기능들로 구성되어 복합연산이라고 한다.

D와 I는 트리의 구조를 변경하므로 노드들의 경로들에 영향을 줄 수 있다. 가령 Fig. 1의 트리에서 D(/A(1)/B(1))가 실행되면 삭제 전의 경로 /A(1)/C(2)는 더 이상 유효하지 않게 되므로, 일반적으로 편집스크립트의 실행 시 편집연산의 노드에 원본트리의 경로를 사용하는 것은 적절하지 않다.

그러나 특정 조건의 편집스크립트의 편집연산들에 원본트리 또는 결과트리의 노드들의 경로가 사용될 수 있다. 일련의 D연산들이 트리의 노드들을 DFS(Depth-First Search)의 역순으로 삭제할 경우, 먼저 처리된 D는 나중에 실행될 D의 대상 노드 경로에는 아무런 영향을 주지 않는다. 이는 DFS의 역순의 특징 때문으로 가령 [D(/A(1)/C(2)), D(/A(1)/B(1))]의 경우, D(/A(1)/C(2))의 실행 후에도 /A(1)/B(1)는 유효하다. 따라서 일련의 D연산들이 삭제 대상 노드들을 기준으로 DFS의 역순으로 구성되어 있는 편집스크립트의 경우, D연산들에는 원본트리에서의 경로의 사용이 가능하다.

일련의 I연산들에도 유사하게 적용된다. 일련의 I연산들이 삽입 위치의 DFS 순으로 구성된 편집스크립트의 경우, 먼저 실행된 I로 인한 변화를 전제로 나중에 실행될 I의 노드 경로가 표현되므로 모든 I의 노드 경로는 모든 I들의 실행 결과로 만들어진 최종 결과트리의 노드 경로와 동일하므로 결과트리의 노드 경로들이 사용될 수 있다. Fig.1의 트리에서 DFS의 역순으로 노드들을 삭제하면 삭제 과정에 남아있는 노드들의 경로는 원본 트리의 경로와 동일하며, 마찬가지로 DFS 순으로 루트 노드부터 삽입하기 시작한다면 I의 노드 경로는 결과트리의 노드 경로와 동일하다.

2.2 X-treeESgen의 개요, 특징 및 문제점

Table 1의 편집연산들은 세 가지 유형으로 분류하는데, 트리 구조를 변화시키지 않으면서 값만을 수정하는 연산들은 '값-변화형', 노드의 삭제 관련 연산들은 '트리-축소형', 그리고 삽입 관련 연산들은 '트리-확장형'이라 한다[17]. Table 1의 U은 값-변화형, D는 트리-축소형, I와 C는 트리-확장형에 속하며, M은 삭제와 붙여넣기로 구성되어 트리-축소형이면서 트리-확장형에 속한다.

2.2.1 기본 편집연산(U, D, I)기반의 편집스크립트의 특징

U, D, I의 기본 연산들로 구성된 편집스크립트

의 생성방법은 비교적 간단하다. X-treeESgen은 X-treeDiff+에서 대응된 원본트리와 결과트리의 노드들을 방문하면서 노드의 대응 유형에 따라 편집연산들을 생성하는데, 우선 원본트리를 DFS 순으로 방문하면서 값-변화형의 편집연산들을 추출한 후, 원본트리를 다시 DFS역순으로 방문하면서 트리-축소형의 편집연산들을 생성하고 마지막으로 결과트리를 DFS순으로 방문하면서 트리-확장형의 연산들을 생성한다. 따라서 편집스크립트는 값-변화형 연산들, 트리-축소형 연산들, 그리고 트리-확장형 연산들의 순으로 구성된다[17]. 이렇게 생성된 편집스크립트는 다음과 같은 특징을 갖는다.

값-변화형 연산들은 트리 구조를 변화시키지 않아 나중에 실행될 편집연산들의 노드 경로에 영향을 주지 않는다. 트리-축소형 연산들은 원본트리에 대해 DFS역순으로 실행하면서 노드들을 제거한다. 트리-확장형 연산들은 모든 트리-축소형 연산들이 실행된, 즉 모든 삭제 작업이 이루어진 트리에 대해 DFS순으로 실행되어 최종 결과트리를 생성한다. 이때 트리-축소형 연산들의 DFS역순의 실행 순서와 트리-확장형 연산들의 DFS 순의 실행순서는 매우 중요하다. 이는 트리-축소형 연산들의 노드 경로들은 원본트리에서의 경로가, 트리-확장형 연산들의 노드 경로들은 결과트리에서의 경로가 사용된다. 따라서 편집스크립트는 순차적인 단일 패스의 실행으로 충분하며, 실행 시 트리의 전체적인 편집 과정이 단조적인 트리의 감소-증가의 구조, 즉 트리의 감소단계 후 트리의 확장단계로 나타난다. 정리하면, X-treeESgen이 생성하는 편집스크립트가 U, D, I만으로 구성되면 단조적인 트리 축소-확장 구조로 단일 패스로 처리가능하다.

2.2.2 X-treeESgen에서 복합연산(C,M) 표현

C(sn, tn, tp, i_ord)는 노드 sn의 서브트리를 tn의 tp번째 자식으로 복사하는 연산으로 복사와 붙여넣기로 구성된다. 이는 대응 과정에서 원본트리의 sn가 결과트리의 다수의 노드들로 대응될 때 생성되며[17], i_ord 속성은 편집스크립트의 트리 확장형 연산들(I, C, M) 내에서 실행 순서를 의미한다. C는 붙여넣기 시점(즉, i_ord의 순서)에 복사될(to copy) 노드(sn)를 접근해야 하는데, 편집스크립트 내에서 C보다 실행 순서가 빠른 연산들이 sn의 경로를 변경시킬 수 있어 C의 붙여넣기 시점에 sn의 경로 식별이 불가능하다. 따라서 XtreeESgen에서는 C는 값 변화형 연산들 생성 후에 위치시키며 sn에 원본트리의 경로를 사용한다.

M(sn, tn, tp, i_ord)도 유사하게 삭제와 붙여넣기로 구성되는데, M의 삭제는 트리의 축소단계에서, M의 붙여넣기는 트리의 확장단계에서 이루어진다. M은 편집스크립트 내에서 이의 삭제의 실행 순서에 위치하고, M의 붙여넣기의 실행 순서는 i_ord 속성을 통해 표현되며 트리 확장형 연산들 내에서의 순서를 의미한다. 참고로 트리 확장형 연산인 I의 i_ord 속성도 트리 확장형 연산들 내에서의 실행 순서를 나타낸다.

2.2.3 XtreeESgen의 편집스크립트의 처리 방법, 특징 및 문제점

편집스크립트가 C와 M을 포함하면 단일 패스의 순차적 실행은 불가능하다. C는 붙여넣기 시점에 sn의 경로를 확인할 수 없고 M은 삭제와 붙여넣기가 서로 다른 시점에 실행되기 때문이다. 따라서 XtreeESgen는 다음의 2단계 패스의 실행방법을 전제로 편집스크립트를 생성한다.

2단계 패스의 실행 방법은 다음과 같다. 첫 번째 패스에서는 모든 C에 대해 복사 내용(서브트리)을, 그리고 모든 M에 대해서는 삭제 내용(서브트리)을 저장한다. 두 번째 패스에서는 값-변화형 연산(U)들과 트리-축소형 연산(D, M의 삭제)들을 순차적으로 실행하고 트리-확장형 연산(I, C와 M의 붙여넣기)들은 i_ord 필드 값의 순서대로 실행한다. 이때 트리-확장 단계에서 C와 M은 첫 번째 패스에서 저장했던 내용들의 붙여넣기 작업을 실행한다.

Fig. 2에서는 편집스크립트, 그리고 원본트리와 2-패스 실행의 결과인 결과트리를 제시한다. 첫 번째 패스에서는 C의 /A(1)/B(1)/D(1)의 노드 D와 M의 /A(1)/B(1)이 나타내는 서브트리를 저장하고, 두 번째 패스에서는 D, M의 삭제, I, M의

[C(/A(1)/B(1)/D(1),/A(1),4,3), D(/A(1)/B(3)), M(/A(1)/B(1),/A(1),3,2), I('G',/A(1),1,1)]

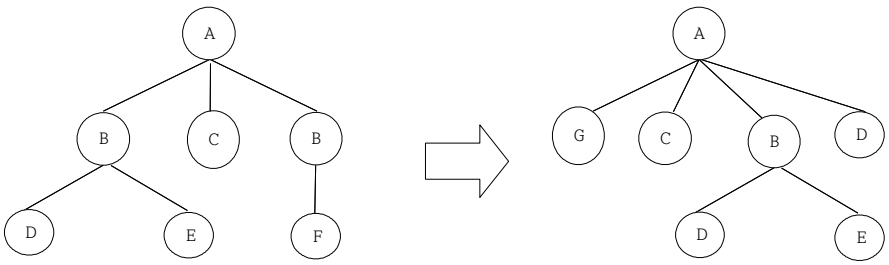

Fig. 2 Two step pass-based edit script and its application to the source tree

삽입, C의 붙여넣기 순으로 실행한다. 실행 결과는 우측 결과트리로 표현되는데, 실행 과정은 이해하기 쉽지 않다. 3절에서는 이를 해결하기 위해 순서교환을 통한 편집스크립트 조정 방법을 제안한다.

3. 트리구조 문서에 대한 U, D, I 연산의 순서교환 방법

본 절에서는 트리 관련 주요 함수 및 연산자들을 정의하고 U, D, I 연산들을 전제로 삭제 또는 삽입에 의해 영향 받는 노드들의 범위, 그리고 편집연산의 순서교환에 대해 소개한다.

3.1 트리 관련 주요 함수 및 연산자

임의의 노드에 대해 경로의 깊이 계산, 형제 순서, 부모 노드의 식별을 위해 depth, pos, parent 함수들, 그리고 tPath 관련 함수와 연산자들을 정의한다. n, m을 트리의 임의의 노드라 할 때, depth(n)은 n의 경로의 깊이를, parent(n)은 n의 부모의 경로를, 그리고 pos(n, d)는 경로 n에서 깊이 d의 위치의 인덱스, 즉 깊이 d의 노드가 몇 번째 자식인지를 계산한다. 예를 들어 Fig. 1의 트리에 대해 depth(/A(1)/B(3)/F(1)) = 3, parent(/A(1)/B(3)/F(1))=/A(1)/B(3), pos(/A(1)/B(1)/E(2),3) = 2, pos(/A(1)/B(1)/E(2), 2) = 1이다.

tPath(n)은 n에 대한 경로(tPath)를 문자열로 반환한다. $<$, $\leqslant$는 문자열을 비교하는 연산자로 임의의 문자열 str1과 str2에 대해 str1 $<$ str2는 str1이 str2의 prefix일 때 true를, 아니면 false를 반환하고, str1 $\leqslant$ str2는 str1 $<$ str2이거나 str1 = str2일 때 true 아니면 false를 반환한다. 이들은 두 노드들의 경로를 비교할 때 사용하는데 tPath(m) $<$ tPath(n)이면 m의 경로가 n의 경로 앞부분과 동일함을 나타내며 이는 m은 n의 조상 노드, n은 m의 자손임을 의미한다. tPath(m) $\leqslant$ tPath(n)의 경우는 위의 경우를 포함하면서 두 노드의 경로가 동일한 경우도 포함한다. 논문에서는 표기의 편의 상 tPath(n)을 직접 n이라 표현한다. 예를 들어 tPath(m) $<$ tPath(n)은 m $<$ n 또는 m $\leqslant$ parent(n)으로 표기된다.

3.2 삭제/삽입에 영향받는 트리의 노드 범위

본 절에서는 D와 I의 실행 시 변화되는 트리의 범위에 대해 분석한다. Fig.1의 트리에서 D(/A(1)/C(2))의 실행은 첫 번째 B와 이의 자손 노드들의 경로에는 아무런 영향이 없으나, 두 번째 B의 경로(/A(1)/B(3))나 F의 경로(/A(1)/B(3)/F(1))는 /A(1)/B(2), /A(1)/B(2)/F(1)으로 변경된다. 이는 경로 표현에 형제 노드들의 순서를 사용하고 있기 때문으로 C의 삭제로 영향을 받는 노드들은 C보다 형제 순서가 늦은 형제 노드들과 이들의 자손 노드들이다. 노드나 서브트리가 삽입될 경우도 위의 예제를 거꾸로 적용하면 이해할 수 있다. 노드 C가 없다는 가정하에 I('C',/A(1),2,_)가 실행되면 경로에 영향 받는 노드들은 C보다 순서가 늦은 형제 노드들과 이들의 자손 노드들이다. 이러한 관찰로부터 다

음의 규칙들이 추론된다.

규칙1 : 트리에 D(n)이 실행되면, n보다 순서가 뒤인 형제노드들과 이들의 자손 노드들의 경로가 변경된다. 삭제 후 n보다 순서가 뒤인 형제노드들의 형제순서는 1만큼 감소된다.

규칙1은 삭제에 의해 영향 받는 노드들의 범위와 조정방법을 설명하는데 D(n)에 의해 영향 받는 노드(m)은 다음의 조건을 만족한다.

조건1: parent(n) < m ∧
pos(n,depth(n))<pos(m, depth(n))

위의 조건은 m이 n보다 순서가 뒤인 형제노드인지 또는 이의 자손 노드인지를 식별한다. 조건1을 통해 n의 삭제 시 경로 조정이 필요한 노드를 확인할 수 있다. Fig.1에서 /A(1)/B(3)/F(1)의 경로는 D(/A(1)/B(1))에 의해 영향 받는 노드들의 범위에 속하므로 삭제 후 /A(1)/B(2)/F(1)로 조정된다.

규칙2 : 트리에 대한 I(c, n, p, _)은 노드 n의 p번째 자식으로 c를 삽입한다. I의 실행 시 n의 자식들에 대해 자식순서가 p와 같거나 순서가 뒤인 자식노드들과 이들의 자손노드들의 경로가 변경된다. 삽입 후 p보다 형제순서가 같거나 뒤인 형제노드들의 형제 순서는 1만큼 증가된다.

규칙2는 삽입에 의해 영향 받는 노드들의 범위와 조정방법을 설명하며 I(_, n, p, _)에 영향 받는 노드(m)은 다음의 조건을 만족한다.

조건2: n < m ∧ p ≦ pos(m , depth(n)+1)

조건2는 m이 I(c, n, p, _)에 의해 삽입될 노드보다 순서가 뒤인 형제노드인지 또는 이의 자손 노드인지를 식별한다. Fig.1에서 예를 들면, I(..., /A(1), 2, _)에 의해 삽입될 노드보다 /A(1)/C(2)와 /A(1)/B(3)는 형제순서가 뒤이고, /A(1)/B(3)/F(1)는 형제순서가 뒤인 노드의 자손노드이므로, 삽입 후 이들은 각각 /A(1)/C(3), /A(1)/B(4), /A(1)/B(4)/F(1)로 경로 조정이 이루어진다. 한편 I(c, n, p, _)에 의해 삽입될 노드를 x로 표현하면 n = parent(x), p = pos(x, depth(x))이고 이때 (조건2)을 x의 관점에 다시 표현하면 다음과 같다.

조건3: parent(x) < m ∧
pos(x, depth(x)) ≦ pos(m, depth(x))

위의 규칙1과 규칙2는 경로(tPath)의 정의로부터 쉽게 도출된다.

3.3 U, D, I 기반의 인접한 편집연산들의 순서 교환

본 절에서는 편집스크립트 내의 인접한 두 연산의 교환 시 연산의 노드 경로의 조정이 필요한 조건과 조정방법에 대해 설명한다. 원본트리 T_0와 U,D,I로 구성되어 단일 패스로 순차적인 실행이 가능한 편집스크립트 [o_1, o_2, ..., o_{n-1}, o_n]를 가정하자. T_0에 대해 이 편집스크립트를 실행하면 o_1은 T_0를 T_1으로, o_2는 T_1를 T_2로, .., o_n은 T_{n-1}을 결과트리 T_n로 변화시킨다. 이때 o_j의 노드의 경로는 T_{j-1}를 전제로 하고, T_{j-1}는 T_0에 o_j이전까지의 편집연산들이 순차적으로 적용된 트리를 의미한다. 두 연산의 편집스크립트 [o_1, o_2]로 단순화하면, o_1의 경로는 T_0을, o_2의 경로는 T_1을 통해 표현되며 T_1은 T_0에 o_1이 적용된 트리이다. 편집연산을 트리에 대한 함수로 나타내면, $T_1 = o_1(T_0)$, $T_2 = o_2(T_1) = o_2(o_1(T_0))$로 표현된다. 함수 개념을 편집스크립트로 확장하면 $T_2 = [o_1, o_2](T_0)$로 표현된다. 본 절의 내용은 $[o_1, o_2](T_0) = [o_2', o_1'](T_0) = T_2$을 만족할 o_1과 o_2의 경로의 조건과 경로 조정 방법을 설명하는 것이다. 단, o_1'와 o_2'는 경로 조정된 연산들을 의미한다.

Table 2, 3, 4에서 o_1은 T_0를 전제로, o_2는 $o_1(T_0)$을 전제로 경로가 표현되는데, 조건은 순서교환 시 조정이 필요한 경우를, 조정방법은 순서교환 후 경로의 조정방법을 의미한다. Table 2는 o_1이 U(n_1, _)인 경우로 U는 노드의 값만을 갱신할 뿐 트리를 변화시키지 않는다. 따라서 순서교환으로 인한 U의 실행 취소의 효과는 후속연산 o_2의 경로에 영향을 주지 않는다. 그러나 o_2가 먼

Table 2 Switching two adjacent edit operations in [U, D], [U, I], or [U, U].

$o_1 = U(n_1, \_)$		
o_2	Condition of tPaths in o_1 and o_2	Adjustment
$D(n_2)$	$parent(n_2) < n_1 \wedge pos(n_2, depth(n_2)) < pos(n_1, depth(n_2))$	$DecPos(n_1, depth(n_2))$
	$n_2 \prec n_1$	conflict, stop switching
$I(\_, n_2, p_2)$	$n_2 \prec n_1 \wedge p_2 \leqq pos(n_1, depth(n_2)+1)$	$IncPos(n_1, depth(n_2)+1))$
$U(n_2, \_)$	No structural change, so no effect on tPath.	

저 실행될 경우 o_1의 경로 n_1은 조정이 필요할 수 있다. 조건과 조정방법의 식별을 위해 편의상 연산명의 쌍과 번호를 사용한다. 가령 UD1은 Table 2에서 U의 D에 대한 첫 번째 조건과 조정방법을, II4는 Table 4에서 I의 I에 대한 네 번째 조건과 조정방법을 의미한다. 조건에 포함되지 않는 경우는 조정 없이 순서교환이 가능하다.

DecPos(n, d)와 IncPos(n, d)는 경로 조정을 위한 함수로 각각 n에서 깊이 d의 위치의 인덱스를 1만큼 감소 또는 증가시킨다. n이 /A(1)/B(1)/E(2)일 때 DecrPos(n, 3)과 IncPos(n, 2)은 n을 각각 /A(1)/B(1)/E(1)과 /A(1)/B(2)/E(2)로 변경한다. (UD1)은 (조건1)에 해당되는 경우로 n_1의 경로 중 n_2의 형제노드의 인덱스를 1만큼 감소시킨다. (UD2)의 경우는 충돌로 간주하여 순서교환 작업을 중지한다. (UI1)은 순서교환 시 (조건2)에 해당되어 n_1의 경로를 조정한다. (UU1)은 o_2가 $U(n_2, _)$인 경우로 항상 조정 없이 순서교환이 가능하다. Fig.1의 트리에 대해 [U(/A(1)/B(**3**)/F(1),_), D(/A(1)/B(1))]는 순서교환 시 [D(/A(1)/B(1)), U(/A(1)/B(**2**)/F(1),_)]로 변경되며 [U(/A(1)/B(**3**)/F(1),_), I('G',/A(1),3)]은 [I('G',/A(1),3)], U(/A(1)/B(**4**)/F(1),_)]로 조정한다.

o_1이 $D(n_1)$인 경우는 Table 3에 제시된다. o_2가 $D(n_2)$인 경우를 분석하면, o_1의 삭제가 o_2의 경로에 영향을 준 경우(DD1), 그렇지 않으면서 순서교환 시 o_2의 실행이 o_1의 경로에 영향을 주는 경우(DD2), o_2의 삭제가 o_1의 노드 n_1의 조상을 삭제하는 경우(DD3)로 구분된다. (DD1)은 o_2의 경로(n_2)가 o_1의 삭제가 이미 반영한 경우로 순서교환 시 삭제의 영향을 o_2의 경로에서 취소, 즉 n_1의 삽입과 동일하여 (조건3)에 따른 조정이 필요하다. (DD2)는 순서교환 시 n_1이 즉 n_2의 삭제에 의해 영향 받는 경우로 (조건1)에 해당되어

Table 3 Switching two adjacent edit operations in [D, D], [D, I], or [D, U].

$o_1 = D(n_1)$		
o_2	Condition of tPaths in o_1 and o_2	Adjustment
$D(n_2)$	$parent(n_1) < n_2 \wedge pos(n_1, depth(n_1)) \leqq pos(n_2, depth(n_1))$	$IncPos(n_2, depth(n_1))$
	$parent(n_2) < n_1 \wedge pos(n_2, depth(n_2)) < pos(n_1, depth(n_2))$	$DecPos(n_1, depth(n_2))$
	$n_2 \prec n_1$	conflict, stop switching
$I(\_, n_2, p_2)$	$parent(n_1) = n_2 \wedge pos(n_1, depth(n_1)) < p_2$	$p_2 = p_2 + 1$
	$parent(n_1) \prec n_2 \wedge pos(n_1, depth(n_1)) \leqq pos(n_2, depth(n_1))$	$IncPos(n_2, depth(n_1))$
	$n_2 \prec n_1 \wedge p_2 \leqq pos(n_1, depth(n_2)+1)$	$IncPos(n_1, depth(n_2)+1))$
$U(n_2, \_)$	$parent(n_1) < n_2 \wedge pos(n_1, depth(n_1)) \leqq pos(n_2, depth(n_1))$	$IncPos(n_2, depth(n_1))$

Table 4 Switching two adjacent edit operations in [I, I].

$o_1 = I(\_, n_1, p_1)$		
o_2	Condition of tPaths in o_1 and o_2	Adjustment
$I(\_, n_2, p_2)$	$n_1 = n_2$	if $p_1 < p_2$ then p_2 -= 1 else p_1 += 1
	$n_1 < n_2 \wedge p_1 < pos(n_2, depth(n_1)+1)$	$DecPos(n_2, depth(n_1)+1)$
	$n_2 < n_1 \wedge p_2 \leqq pos(n_1, depth(n_2)+1)$	$IncPos(n_1, depth(n_2)+1)$
	$n_1 < n_2 \wedge p_1 = pos(n_2, depth(n_1)+1)$	conflict, stop switching

조정이 필요한 경우다. (DD3)은 순서교환 시 삭제할 대상이 이미 삭제되어 존재하지 않아 순서교환을 할 수 없는 경우다. Fig.1의 트리에 대한 [D(/A(1)/C(2)), D(/A(1)/B(**2**)/F(1))]는 (DD1)에 해당되어 [D(/A(1)/**B(3)**/F(1)), D(/A(1)/C(2))]로 조정된다. 한편 [D(/A(1)/C(2)), D(/A(1)/B(1))]는 (DD2)에 해당되며 순서 교환 시 [D(/A(1)/B(1)), D(/A(1)/**C(1)**)]로 조정된다.

$o_2 = U(n_2,_)$인 경우 즉 (DU1)은, 순서교환 시 n_2에 대해 n_1의 삭제 취소(즉 n_1의 삽입) 효과가 발생하므로 (조건3)에 해당할 때 n_2의 경로를 조정한다. 가령 [D(/A(1)/C(2)), U(/A(1)/B(2)/F(1),_)]는 [U(/A(1)/**B(3)**/F(1),_), D(/A(1)/C(2))]로 조정된다. 순서교환 후 U는 후속연산 o_1의 경로에 아무런 영향을 주지 않는다.

$o_2 = I(_, n_2, p_2)$일 때, (DI1, DI2)는 n_1의 삭제가 n_2의 경로에 이미 반영된 경우들이고, (DI3)은 그렇지는 않으나 순서교환 시 o_2의 실행이 n_1의 경로에 영향을 주는 경우다. (DI1, DI2)는 순서교환 시 n_1의 삭제 취소가 n_2에 영향을 주는 경우로 (DI1)은 o_2로 인해 삽입될 노드가 n_1과 형제인 경우, (DI2)는 n_1과 삽입될 노드가 n_1의 형제의 자손인 경우의 조건과 조정방법을 제시한다. (DI3)은 순서교환 후 (조건3)에 해당하면, o_2의 노드 삽입으로 인한 n_1의 조정이 필요함을 의미한다. 한편, 순서교환으로 인한 n_1의 삽입 위치와 o_2의 노드 삽입 위치가 동일한 경우는 (DI3)에서 처리하고 (DI1)에서는 형제 순서에 대한 동등비교 조건을 포함시키지 않는다. [D,I]와 [D,U]인 경우는 순서교환 시 충돌은 발생할 수 없음을 참고하시오. Fig.1의 트리에 대한[D(/A(1)/C(2)), I("G",/A(1),3)]는 (DI1)에 해당되어 [I("G",/A(1),**4**), D(/A(1)/C(2))]로 조정되며, [D(/A(1)/C(2)), I("G",/A(1)/B(2)/F(1),1)]는 (DI2)가 적용되어 [I("G",/A(1)/**B(3)**/F(1),1), D(/A(1)/C(2))]로 조정된다. [D(/A(1)/B(3)/F(1)), I("G",/A(1),3)]은 (DI3)에 해당되며 순서를 교환하면 [I("G",/A(1),3), D(/A(1)/**B(4)**/F(1))]로 조정된다.

Table 4는 o_1은 $I(\_, n_1, p_1)$, o_2는 $I(_, n_2, p_2)$인 경우를 분석한다. (II1)은 두 연산이 동일 부모에 삽입하는 경우인데, 순서교환 시 o_1의 삽입 취소 효과로 인한 p_2에 대한 조건과 조정방법, 그리고 o_2가 먼저 실행됨으로 인한 p_1의 조건과 조정방법을 보인다. 가령 [I('G',/A(1), **2**), I('K',/A(1),2)]

Table 5 Transform composite operations(C, M) into simple operations.

Composite Operation	Simple Operation	Description
C(sn, tn, tp, i_ord)	C_c(sn)	Copy(save) the subtree rooted at sn.
	C_p(tn, tp)	Paste the copied subtree of C_c as the tp^{th} child of tn.
M(sn, tn, tp, i_ord)	M_c(sn)	Cut and save the subtree rooted at sn.
	M_p(tn, tp)	Paste the saved subtree of M_c as the tp^{th} child of tn.

Table 6 Switching two adjacent edit operations in [M_c, _], or [C_c, _].

		o_1	
		M_c(n_1)	C_c(n_1)
o_2	D(n_2), M_c(n_2)	DD1, DD2, DD3 in Table 3	UD1, UD2 in Table 2
	I(_, n_2, p_2), M_p(n_2, p_2), C_p(n_2, p_2),	DI1, DI2, DI3 in Table 3	UI1 in Table 2
	U(n_2), C_c(n_2)	DU1 in Table 3	UU1 in Table 2

는 [I('K',/A(1), 2), I('G',/A(1), 3)]으로 조정된다. (II2)는 o_1의 삽입 취소 효과가 n_2에 영향을 주는 조건과 조정방법을 보인다. [I('G',/A(1),2), I('K',/A(1)/C(3),1)]은 순서교환 시 [I('K',/A(1)/C(2),1), I('G',/A(1),2)]로 조정된다. (II3)은 순서교환 시 o_2의 삽입 효과가 n_1에 영향을 주는 조건과 조정방법을 나타낸다. [I('G',/A(1)/C(2),1), I('K',/A(1),2)]는 순서교환 시 [I('K',/A(1),2), I('G',/A(1)/C(3), 1)]로 조정된다. (II4)는 o_1에 의해 삽입된 노드(서브트리)에 o_2가 삽입하는 경우로 순서교환 시 삽입이 불가능하기 때문에 순서교환을 불허한다.

4. 비단조적 증감 구조의 단일 패스 편집 스크립트 변환 알고리즘

4.1. 단일 패스 편집스크립트 변환 알고리즘

본 절에서는 XtreeESgen이 생성한 편집스크립트를 단일 패스로 처리하기 위한 편집스크립트 변환 알고리즘을 소개한다. 이의 핵심은 복합연산인 C와 M을 단순연산들로 분리하고 편집연산들의 순서교환을 통해 단일 패스로 처리할 수 있는 C^s와 M^s으로 재구성하는데 있다. C^s와 M^s는 C와 M에서 i_ord가 삭제된 형태를 복사와 붙여넣기, 잘라내기와 붙여넣기가 한 번에 실행되는 연산이다. Table 5에서 제시한대로 복합연산 C(sn, tn, tp, i_ord)는 복사작업(copy)의 C_c(sn)과 붙여넣기(paste)의 C_p(tn, tp)로, 그리고 M(sn, tn, tp, i_ord)은 잘라내기(cut)의 M_c(sn)과 붙여넣기(paste)의 M_p(tn, tp)로 분리된다.

단일 패스 편집스크립트 변환 알고리즘은 다음의 세 세부단계로 진행된다.

(1) 복합연산의 단순연산으로 분리
(2) 편집연산의 순서교환으로 C^s와 M^s을 추출
(3) C^s와 M^s으로 추출되지 않은 C와 M의 구성 연산들을 D와 I로 전환

(1) 복합연산을 단순연산으로 분리: 편집스크립트에서 C와 M을 찾아 다음과 같이 변경한다. C는 C_c와 C_p로 분리하는데, C_c와 C_p의 ID는 C와 동일하게 설정하며 편집스크립트 내에서 C_c는 '값 변화형' 연산들의 뒤의 원래 위치에, 그리고 C_p는 '트리-확장형'연산들의 i_ord 순서에 위치한다. M도 M_c와 M_p로 분리하면서 동일한 ID를 갖도록 설정하고 편집스크립트 내에서 M_c는 M의 원래 순서에, M_p는 '트리-확장형'연산들의 i_ord 순서에 위치한다. 복합연산들을 단순연산으로 분리하면 Fig.2의 편집스크립트는 [C_c(A(1)/B(1)/D(1)), D(/A(1)/B(3)), M_c(/A(1)/B(1)), I('G',/A(1),1), M_p(/A(1),3), C_p(/A(1),4)]로 변환된다. i_ord 속성이 더 이상 사용되지 않음에 주목하시오.

(2) 편집연산의 순서교환을 통한 C^s와 M^s 추출: Table 6의 M_c와 C_c의 후속연산과의 순서교환의 조건과 경로조정 방법, 그리고 Table 7의 M_p와 C_p의 직전연산과의 순서교환의 조건과 경로조정 방법을 사용하여 이들 연산에 대해 동일 ID의 연산이 인접할 때까지 순서교환을

한 후 인접한 동일한 ID의 편집연산들에 대해 C^s 또는 M^s을 추출한다. 편집연산들의 순서교환 과정은 다음과 같이 진행된다.

실행 순서가 가장 늦은 M_c부터 이의 후속연산과의 다음의 순서교환 작업을 진행한다. 이 M_c에 대응하는(동일 ID의) 후속연산 M_p를 만나거나 후속연산과 순서교환이 불가능하면 순서교환을 중단하고, 순서교환이 가능하면 필요한 경로조정 후 순서교환을 한다. 순서교환 후 순서교환된 M_c와 그 다음 후속연산과의 위의 순서교환 작업을 반복 시도한다. 순서교환이 중단되면 그 다음으로 실행 순서가 늦은 M_c에 대해 위의 작업을 반복하여 모든 M_c에 대해 순서교환 작업을 완료한다.

모든 M_c에 대한 순서교환 작업 완료 후 M_p와 C_p에 대해 순서교환 작업을 진행한다. 이들 중 실행 순서가 빠른 M_p 또는 C_p부터 직전연산과의 순서교환을 시작하는데 이 과정은 순서교환의 방향이 반대이고 대응연산이 M_c 또는 C_c임을 제외하고는 위의 M_c의 순서교환 작업과 동일하다. 순서교환이 중단되면 그 다음의 실행순서가 빠른 M_p 또는 C_p에 대해 동일한 작업을 진행한다. 모든 C_p와 M_p의 처리 후 C_c에 대해 순서교환을 완료한다. C_c의 경우, 실행 순서가 가장 늦은 C_c부터 순서교환을 시작하며 과정은 M_c의 경우와 유사하다.

모든 순서교환이 완료된 편집스크립트에 대해 인접한 동일 ID의 M_c(sn)와 M_p(tn, tp), 또는 인접한 동일 ID의 C_c(sn)와 C_p(tn, tp)가 있으면 이를 각각 M^s(sn, tn, tp) 또는 C^s(sn, tn, tp)로 통합한다.

(3) 추출되지 않은 C와 M의 구성 연산들을 D와 I로 전환 : 모든 C^s와 M^s이 추출된 후의 편집스크립트에는 C_c와 C_p 그리고 M_c와 M_p가 남아 있을 수 있다. 이들은 동일 ID이지만 인접하지 않아 위의 마지막 단계에서 처리되지 않은 경우로 다음과 같이 처리한다. 동일 ID의 C_c와 C_p에 대해 C_c의 복사내용을 C_p의 순서에서 I로 대치하고 기존의 C_c와 C_p는 삭제한다. 동일 ID의 M_c와 M_p에 대해 M_c는 D로 대치하고 이동대상은 M_p의 위치에서 I으로 삽입하고 기존의 M_c와 M_p는 삭제한다.

4.2 인접한 편집연산들의 순서교환의 조건 및 경로 조정 방법

위의 단일 패스 편집스크립트 변환 알고리즘은 M_c, C_c와 이의 후속연산과의 순서교환과 경로조정 방법을, M_p, C_p와 이의 직전연산과의 순서교환과 경로조정 방법을 전제로 한다. Table 6은 M_c, C_c에 대한 후속연산과의 순서교환 조건과 조정 방법을, Table 7은 M_p, C_p에 대한 직전연산과의 순서교환 조건과 경로조정 방법을 제시한다. 트리의 구조 변화 관점에서, M_c는 잘라내기로 D와 동일한 효과를, C_c는 복사작업으로 트리의 구조에 변화를 주지 않으므로 U와 동일한 효과를, M_p, C_p는 I와 동일한 효과를 갖는다. 따라서 순서교환 대상연산들을 {D, M_c}, {I, M_p, C_p}, {U, C_c}의 세 가지 유형으로 나누어 분석한다.

Table 6에서는 o_1이 M_c 또는 C_c인 경우, 후속연산 o_2이 위의 세 그룹 중 어디에 속하는가에 따라 순서교환 조건과 경로조정 방법을 제시한다. M_c와 후속연산(o_2)과의 순서교환에 대한 내용은 두 번째 열에, C_c에 대한 내용은 세 번째 열에 제시된다. o_1과 o_2가 주어지면 해당 셀의 내용을 참조하여 관련 조건과 조정 방법을 알 수 있다. 가령, Fig.1의 트리에 대한 [M_c(/A(1)/C(2)), D(/A(1)/B(2)/F(1))]는 [M_c, D]인 경우로 Table

Table 7 Switching two adjacent edit operations in [_, M_p] or [_, C_p].

		o_1		
		D(n_1), M_c(n_1)	I(_, n_1, p_1), M_p(n_1, p_1), C_p(n_1 p_1)	U(n_1), C_c(n_1)
o_2	M_p(n_2 p_2), C_p(n_2 p_2)	DI1, DI2, DI3 in Table 3	II1, II2, II3, II4 in Table 4	UI1 in Table 2

3의 DD1, DD2, DD3이 적용 가능함을 의미한다. 이 경우는 Table 3의 DD1에 해당되어 순서 교환 시 [D(/A(1)/**B(3)**/F(1)), M_c(/A(1)/C(2))]로 조정된다. Table 7에서는 o_2가 M_p 또는 C_p인 경우 직전연산 o_1과의 순서교환 조건과 경로조정 방법이 제시된다. 사용법을 예를 들면, [I('G',/A(1),2), M_p(/A(1)/C(3),1)]은 II1, II2, II3, II4 중 I2가 적용 가능하여 순서교환 시 [M_p(1)/**C(2)**,1), I('G',/A(1),2)]로 조정된다.

4.3 단일 패스 편집스크립트 변환 알고리즘의 예제 제시와 분석

단일 패스 편집스크립트 변환 알고리즘의 적용 예제와 알고리즘의 시간복잡도 분석과 이의 활용에 대해 설명한다. Fig. 2의 편집스크립트에 대해 단일 패스 편집스크립트 변환 알고리즘을 적용한다. 우선 복합연산들을 단순연산들로 분리하면, [C_c(A(1)/B(1)/D(1)), D(/A(1)/B(3)), M_c(/A(1)/B(1)), I('G',/A(1),1), M_p(/A(1),3), C_p(/A(1),4)]로 표현된다. 그리고 C^s와 M^s의 추출을 위해 Table 6과 7의 내용을 기반으로 순서교환 작업을 진행한다.

(1) M_c의 순서교환

DI3에 의해 [M_c(/A(1)/B(1)), I('G',/A(1),1)] → [I('G',/A(1),1), M_c(/A(1)/B(2))]. 따라서 다음과 같이 변경. [C_c(A(1)/B(1)/D(1)), D(/A(1)/B(3)), **I('G',/A(1),1),M_c(/A(1)/B(2))**, M_p(/A(1),3), C_p(/A(1),4)]. 그리고 동일한 ID의 M_c와 M_p가 인접하므로 멈춤.

(2) M_p와 C_p의 순서교환

(a) M_p(/A(1),3)부터 시작하나 직전연산과 동일한 ID의 연산이므로 멈추고 다음으로 진행.

(b) C_p(/A(1),4)로 시작. III에 의해 [M_p(/A(1),3), C_p(/A(1),4)] → [C_p(/A(1),3), M_p(/A(1),3)]. 전체 스크립트는 다음과 같이 수정됨. [C_c(A(1)/B(1)/D(1)), D(/A(1)/B(3)), I('G',/A(1),1), M_c(/A(1)/B(2)), **C_p(/A(1),3)**, **M_p(/A(1),3)**].

(c) DI1에 의해 [M_c(/A(1)/B(2)),C_p(/A(1),3)] → [C_p(/A(1),4), M_c(/A(1)/B(2))]. 전체는 [C_c(A(1)/B(1)/D(1)),D(/A(1)/B(3)), I('G',/A(1),1), **C_p(/A(1),4), M_c(/A(1)/B(2))**, M_p(/A(1),3)]로 수정됨.

(d) [I('G',/A(1),1), C_p(/A(1),4)]은 II1이 적용되어 [C_p(/A(1),3), I('G',/A(1),1)]로 조정되고 전체는 [C_c(A(1)/B(1)/D(1)), D(/A(1)/B(3)), **C_p(/A(1),3), I('G',/A(1),1)**, M_c(/A(1)/B(2)), M_p(/A(1),3)]로 수정됨.

(e) [D(/A(1)/B(3)), C_p(/A(1),3)]는 DI3을 만족하므로 [C_p(/A(1),3), D(/A(1)/B(4))]로 수정. [C_c(A(1)/B(1)/D(1)),**C_p(/A(1),3), D(/A(1)/B(4))**, I('G',/A(1),1), M_c(/A(1)/B(2)), M_p(/A(1),3)]

(3) C_c와의 순서교환: 동일한 ID 연산이 인접해 있으므로 순서교환을 멈춘다.

(4) C^s와 M^s의 추출

[C^s(A(1)/B(1)/D(1),/A(1),3), D(/A(1)/B(4)), I('G',/A(1),1), M^s(/A(1)/B(2),/A(1),3)]로 조정. 끝으로 추출되지 않은 C와 M의 구성연산이 없으므로 종료한다.

단일 패스 편집스크립트 변환 알고리즘의 시간복잡도는 m개의 연산에 대해 다음과 같이 분석된다. (1)은 C, M을 찾아 C_c와 M_c는 원래의 위치에, 그리고 C_p와 M_p는 이들의 i_ord의 위치에 배치하는 작업으로 ID 딕셔너리를 사용하면 두 번의 패스로 가능하고, (3)은 한 번의 패스를 통해 처리 가능하다. (2)는 M_c, M_p, C_p, C_c 유형의 연산들에 대해 인접 연산과의 교환 작업인데 각 경우에 대해 스크립트 전체에 대한 패스를 가정하고 모든 연산들이 이들 유형으로 정해졌다고 가정할 경우 $O(m^2)$이므로 (1),(2),(3) 전체에 대해 $O(m^2)$로 표현된다. 그러나 M과 C의 연산의 수가 10% 이하로 추정되며 이들 각각에 대해 평균 편집스크립트 길이의 반에 대한 패스가 발생하므로 실제로 비용은 크지 않다. 대응된 트리들에 대해 두 트리의 총 노드 수를 n이라 할 때 X-treeESgen의 시간 복잡도는 O(n)이며 이는 실험을 통해서도 확인되었다[17]. 본 알고리즘이 X-treeESgen에 대한 추가되어도 총 노드의 수에 비해 연산의 수는 매우 적은 수치로 실험 결과에는 영향이 없다.

4절의 편집스크립트 변환 알고리즘이 대상으로 하는 편집스크립트는 모든 형태의 편집스크립트가 아니라 XtreeESgen의 편집스크립트로 이는 다음 특징들을 전제로 한다. (1) 원본트리에 항상 적용 가능하고, (2) 단조적인 트리 증감 구조의 2단계 패스 실행을 전제로 하며, (3) 원본트리에 적용 시 결과트리를 생성한다. 한편, 알고리즘 적용 후 변환된 편집스크립트도 (1)과 (3)을 만족하나 편집연산들의 순서 교환으로 인해 단조적인 증감구조는 만족하지 않는다. 한편, 3절의 내용은 X-treeESgen의 편집스크립트에만 해당되는 내용은 아니다. 이는 tPath로 노드의 경로가 표현 가능한 경우, 기본연산들을 지원하는 편집스크립트에 적용 가능하며 이를 통해 동일한 작업의 편집스크립트들을 생성할 수 있어 편집스크립트의 효율성 개선 등의 작업이 가능하다.

5. 결 론

최근 XML은 모바일 플랫폼을 포함한 다양한 플랫폼에서 점차 그 사용이 확산되고 있으며 XML 문서들을 기존의 RDB에서 효과적으로 표현하고 관리하는 연구가 진행되고 있다[19-20]. 이러한 추세는 XML 유사 언어의 문서의 활용에도 확산되고 있어, 트리구조의 문서들에 대한 버전 관리는 더욱 중요해지고 있다.

본 논문에서는 협업 기반의 트리구조의 문서 작업 시 버전 관리의 기반이 되는 편집스크립트의 실행 과정의 특성을 이해하기 위한 문서 모델을 제안하고 이를 통해 편집연산의 실행에 따른 트리 변화와 이 변화가 후속연산에 끼치는 영향을 분석했다. 그리고 이를 통해 편집스크립트 내에서 연산들의 순서 교환의 조건과 조정 방법을 제시했다. 편집스크립트는 diff 알고리즘들의 적용 시, 또는 사용자의 문서 편집 과정에서 생성되는데 편집스크립트가 tPath를 사용하면 본 연구 결과의 적용이 가능하다. 제시한 연구 결과는 약간의 확장을 통해 편집스크립트의 성능 개선, 편집스크립트의 병합 등에 적용 가능한데, 본 논문에서는 이를 활용하여 복합연산들이 포함된 X-treeESgen의 편집스크립트를 단일 패스 편집스크립트로 변환하는 알고리즘을 제시하였다. 현재 다중 사용자들에 의한 편집스크립트들의 병합 알고리즘을 개발하고 있고 이에 대한 연구 논문이 준비 중이다.

References

[1] Ronnau, S., Scheffczyk, J. and Borghoff, U., "Towards XML Version Control of Office Documents," Proceedings of ACM Symposium on Document Engineering, pp. 10-19, 2005.

[2] Antila, C., Trevino, J. and Weaver, G. "A Hierarchic Diff Algorithm For Collaborative Music Document Editing," Proceedings of Technologies for Music Notation & Representation, 2017.

[3] Weaver, C. and Smith, S., "XUTools: Unix Commands for Processing Next-Generation Structured Text," Proceedings of Large Installation System Adminstration Conference, 2012.

[4] Ronnau, S. and Borghoff, U., "XCC: Change Control of XML Documents," Computer Science - Research and Development, Vol. 27, Issue 2, pp. 95-111, 2012.

[5] Lee, Suk Kyoon, "Change Detection of Hangul Documents Based on X-treeDiff+," Journal of the Korea Industrial Information Systems Society, Vol. 15, No. 4, pp. 29-37, 2010.

[6] Selkow, S. "The Tree-To-Tree Editing Problem," Information Processing Letters, Vol. 6, No. 6, 1977.
DOI:10.1016/0020-0190(77)90064-3

[7] Zhang, K. and Shasha, D., "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," SIAM Journal of Computing, Vol. 18, No. 6, pp. 1245-1262, 1989.

[8] Cobéna, G., Abiteboul, S. and Marian, A., "Detecting Changes in XML Documents,"

Proceedings of the 18th International Conference on Data Engineering, 2002.
[9] Lee, S. and Kim, D., "X-treeDiff+: Efficient Change Detection Algorithm in XML Documents," Lecture Notes in Computer Science, Vol. 4096, pp. 1037-1046, 2006.
[10] Lee, S. and Kim, D., "Improving Performance of Change Detection Algorithms through the Efficiency of Matching," KIPS Transactions on Software and Data Engineering, Vol. 14, No. 2, pp. 145-156, 2007.
[11] Chawathe, S., "Comparing Hierarchical Data in External Memory," Proceedings of the 25th International Conference on Very Large Data Bases. 1999.
[12] diffxml, http://diffxml.sourceforge.net/ (accessed on Mar. 21th, 2019)
[13] Fontaine, R., "Change Control for XML: Do it right," Proceedings of XML Europe Conference, 2003.
[14] DeltaXML, http://www.deltaxml.com (accessed on Mar. 21th, 2019)
[15] Dohn, H. and Riechle, D., "Fine-grained Change Detection in Structured Text Documents," Proceedings of the 2014 ACM Symposium on Document Engineering, pp. 87-96, 2014.
[16] Autexier, S., "Similarity-Based Diff, Three-Way Diff and Merge," International Journal of Software Informatics, Vol 9, Issue 2, pp 259-277, 2015.
[17] Lee, S., "An Algorithm Generating Edit Scripts for XML Documents," Journal of the Institute of Electronics and Information Engineers: CI, Vol. 48, No. 1, pp. 80-89, 2011.
[18] Kang, J., "A Study on Version Management of Documents with Hierarchical Structure," Master Thesis, Dankook University, 2013.
[19] Kim, S., Jung, S., Kang, Y. and Cho, W., "Mobile Office Construction on a Geotechnical Information System," Journal of the Korea Industrial Information Systems Society, Vol. 15, No. 5, pp. 125-135, 2010.
[20] Woo, W., "A Study on Developing XML Documents and RDB Mapping Using Tag Free XML Development Tools," Journal of the Korea Industrial Information Systems Society, Vol. 11, No. 5, pp. 37-52, 2006.

이 석 균 (Lee SukKyoon)

- 서울대학교 경제학과 학사
- U. of Iowa, 전산학 석사
- 세종대학교 전임강사
- 단국대학교 SW융합대학 소프트웨어학과 교수
- 관심분야 : 데이터 모델, 불완전 정보관리, 시각질의어, 문서의 변화 탐지 및 버전 관리, 기계학습

엄 현 민 (Um HyunMin)

- 단국대학교 공과대학 소프트웨어학과 학사과정
- 관심분야 : 패턴 매칭, 기계학습