KIPS Tr. Software and Data Eng.
Vol.7, No.2 pp.51~62 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2018.7.2.51

Mining High Utility Sequential Patterns Using Sequence Utility Lists

Jong Soo Park†

ABSTRACT

High utility sequential pattern (HUSP) mining has been considered as an important research topic in data mining. Although some algorithms have been proposed for this topic, they incur the problem of producing a large search space for HUSPs. The tighter utility upper bound of a sequence can prune more unpromising patterns early in the search space. In this paper, we propose a sequence expected utility (SEU) as a new utility upper bound of each sequence, which is the maximum expected utility of a sequence and all its descendant sequences. A sequence utility list for each pattern is used as a new data structure to maintain essential information for mining HUSPs. We devise an algorithm, high sequence utility list-span (HSUL-Span), to identify HUSPs by employing SEU. Experimental results on both synthetic and real datasets from different domains show that HSUL-Span generates considerably less candidate patterns and outperforms other algorithms in terms of execution time.

Keywords : High Utility Sequential Pattern Mining, Sequence Utility List, Candidate Pattern Pruning

시퀀스 유틸리티 리스트를 사용하여 높은 유틸리티 순차 패턴 탐사 기법

박 종 수†

요 약

높은 유틸리티 순차 패턴 탐사는 데이터 마이닝에서 중요한 연구 주제로 간주되고 있다. 이 주제에 대해 몇 개의 알고리즘들이 제안되었지만, 그것들은 높은 유틸리티 순차 패턴 탐사의 탐색 공간이 커지는 문제에 부딪히게 된다. 한 시퀀스의 더 엄격한 유틸리티 상한 값은 탐색 공간에서 초기에 유망하지 않은 패턴들을 더 가지치기할 수 있다. 본 논문에서 새로운 유틸리티 상한 값을 제안하는데, 그것은 한 시퀀스와 그 자손 시퀀스들의 최대 예상 유틸리티인 sequence expected utility (SEU)이다. 높은 유틸리티 순차 패턴들을 탐사하는데 필수적인 정보를 유지하기 위해 각 패턴에 대한 시퀀스 유틸리티 리스트를 새로운 자료구조로 사용한다. SEU를 활용하여 높은 유틸리티 순차 패턴들을 찾아내는 알고리즘인 High Sequence Utility List-Span (HSUL-Span)을 제안한다. 서로 다른 영역의 합성 데이터세트와 실제 데이터세트에 대한 실험 결과는 HSUL-Span이 상당히 적은 수의 후보 패턴들을 생성하고 실행 시간 면에서 다른 알고리즘들보다 우수한 것을 보여준다.

키워드 : 높은 유틸리티 순차 패턴 마이닝, 시퀀스 유틸리티 리스트, 후보 패턴 가지치기

1. 서 론

순차 패턴 탐사는 연관 규칙 탐사[1-3]에 이어서 데이터 마이닝에서 중요한 분야로 인식된다. 순차 패턴[4-6]은 여러 항목들이 시간 순서로 주어질 때 지정된 최소 임계값 이상의 지지를 받는 패턴을 탐사하는 것으로 연구가 많이 이루어졌다. 순차 패턴 탐사나 연관 규칙 탐사는 항목들의 존재 유무를 가지고 원하는 패턴을 탐사하지만, 유틸리티 패턴 탐사는 항목들이 가지는 유용성이나 가치가 서로 다른 것을 고려한다. 각 패턴에 속한 항목들의 이익의 합이 그 패턴의 유틸리티가 된다. 사용자가 지정된 최소 임계값 이상의 이익을 가지는 패턴을 탐사하는 높은 유틸리티 패턴 탐사[7-11]가 먼저 연구되고, 그 이후에 높은 유틸리티 순차 패턴 탐사 방법이 연구되고 있다.

높은 유틸리티 순차 패턴(High Utility Sequential Pattern, HUSP)을 탐사하는 문제는 입력으로 시퀀스 데이터베이스가 주어지면 그 데이터베이스에 지정된 최소 임계값 이상의 유틸리티를 가지는 순차 패턴들을 찾는 것이다. 시퀀스 데이터베이스의 각 시퀀스는 여러 개의 항목들로 이루어진 항목집합들이 순차적으로 나열되고, 항목집합들의 각 항목은 몇

※ 이 논문은 2015년도 성신여자대학교 학술연구조성비 지원에 의하여 연구되었음.
† 종신회원 : 성신여자대학교 IT학부 교수
Manuscript Received : October 13, 2017
First Revision : December 6, 2017
Accepted : December 18, 2017
* Corresponding Author : Jong Soo Park(jpark@sungshin.ac.kr)

개가 포함되는 지를 나타내는 내부 유틸리티(internal utility)와 항목 하나에 대한 가치를 나타내는 외부 유틸리티(external utility)를 가지고 있다. 항목들이 대형 마트의 상품들이라면 내부 유틸리티는 그 상품이 팔린 개수를 나타내고, 외부 유틸리티는 그 상품의 판매가나 단위 이익을 나타낸다.

높은 유틸리티 순차 패턴을 탐사하는 것은 다음 두 가지 측면에서 어려운 문제점이 있다. 첫 번째가 빈발 패턴 탐사에서 적용된 downward closure property가 더 이상 적용되지 않아서 한 시퀀스의 유틸리티가 그것의 부분 시퀀스의 유틸리티 보다 크다는 보증도 없고 작다는 보증도 없게 되어 패턴들을 만들어 가는 탐색 공간에서 가지치기를 할 수 없게 된 것이다[7]. 이 문제를 해결하기 위해, 한 시퀀스와 그 시퀀스로 만들어지는 자손 패턴들의 유틸리티의 상한치로 시퀀스-가중치 유틸리티(sequence-weighted utility, SWU)를 적용하여 높은 유틸리티 순차 패턴 탐사에서 탐색 공간을 가지치기하도록 하였다[12, 13]. 한 시퀀스의 SWU가 최소 유틸리티 임계값보다 작으면 그 시퀀스는 더 이상의 패턴 확장을 할 수 없도록 가지치기를 할 수 있다. 그렇지 않으면 그 시퀀스는 높은 유틸리티 순차 패턴의 후보가 되어 자손 패턴들에 대한 탐색이 계속된다. 두 번째 문제는 시퀀스들의 조합으로 이루어지는 후보 높은 유틸리티 순차 패턴들이 거대한 탐색 공간을 만들어 후보 패턴들의 개수가 아주 많아져서 높은 계산 복잡도를 가진 문제가 된다는 것이다.

본 논문에서 높은 유틸리티 순차 패턴을 탐사하고자 제안하는 알고리즘 HSUL-Span은 입력인 시퀀스 데이터베이스의 각 시퀀스에 대한 시퀀스 유틸리티 매트릭스를 기본적인 자료구조로 사용한다. 그리고 패턴들을 탐색해나가는 과정에서 각 패턴에 해당되는 시퀀스에 대하여 새로운 자료구조인 시퀀스 유틸리티 리스트를 제안한다. 기존의 리스트들[14-16]과 다른 새로운 시퀀스 유틸리티 리스트는 해당 시퀀스인 패턴을 포함하고 있는 시퀀스 유틸리티 매트릭스의 정보를 튜플 단위로 저장하고 관리한다. 패턴을 확장해가는 과정에서는 시퀀스 유틸리티 리스트에 있는 튜플들에서 접합될 수 있는 항목들을 찾아낸다. 각 튜플에서 접합의 조건을 만족하는 모든 항목들의 예상 유틸리티를 각 항목별로 저장한다. 이렇게 저장된 항목들 중에서 한 항목의 예상 유틸리티가 사용자-지정 최소임계값 이상의 유틸리티를 가지면 그 항목은 현재 패턴의 접합 항목이 된다. 현재 패턴인 시퀀스에 찾아진 접합 항목을 접합하면 새로운 패턴이 생성되고 그 패턴에 대한 시퀀스 유틸리티 리스트도 만든다. 각 패턴의 시퀀스 유틸리티 리스트에서 해당 패턴의 유틸리티 값이 지정된 최소 유틸리티 보다 크거나 같으면 그 패턴은 높은 유틸리티 순차 패턴으로 찾아진다. 제안된 알고리즘의 특성은 다음과 같다:

1. 항목이나 시퀀스로 표현되는 패턴에 대하여 새로운 자료구조인 시퀀스 유틸리티 리스트를 제안한다. 각 패턴의 시퀀스 유틸리티 리스트에서 패턴의 유틸리티, 최대 예상 유틸리티, 그리고 패턴 확장 시에 필요한 접합 항목들을 계산한다.
2. 탐색 공간에서 가지치기에 사용할 수 있는 유틸리티 상한 값으로 한 시퀀스의 시퀀스 최대 예상 유틸리티(sequence expected utility, SEU)를 정의한다. 한 시퀀스의 SEU가 사용자-지정 최소 유틸리티 임계값 보다 작으면 안전하게 가지치기를 할 수 있음을 보여준다.
3. 제안된 알고리즘인 HSUL-Span의 초기 단계에서 시퀀스 데이터베이스에 속한 항목들 중에서 어떤 항목의 SWU 값이 사용자-지정 최소 유틸리티 임계값 보다 작으면 그 항목을 시퀀스 데이터베이스에서 제외하는 트리밍(trimming) 과정을 수행한다. 그러면 패턴의 접합에 사용되는 후보 항목들의 개수를 최대한 줄이게 되어 후보 패턴들도 줄인다.

나머지 논문의 구성에 대해 설명하면, 2 장에서는 기존 연구에 대한 관련 연구를 설명하고, 3 장에서는 높은 유틸리티 순차 패턴을 탐사하는 과정에서 필요한 개념들을 정의하고 탐사 문제를 정의한다. 4 장에서는 제안하는 알고리즘에 필요한 시퀀스 유틸리티 매트릭스와 시퀀스 유틸리티 리스트에 대해 설명한다. 그리고 제안된 알고리즘을 기술하고, 알고리즘의 함수들에 대하여 상세히 서술한다. 5 장에서는 제안된 알고리즘의 성능 평가를 위한 실험 결과를 기술하고, 6 장에서는 논문의 결론을 맺는다.

2. 관련 연구

높은 유틸리티 순차 패턴 마이닝의 문제를 해결하기 위하여 Ahmed 등[12]은 데이터베이스 스캔 횟수를 줄이고 탐색 공간을 순환적으로 나누어가는 패턴 성장 접근법을 적용하여 후보 패턴을 줄이는 US 알고리즘을 제시하였다. Yin 등[13]은 높은 유틸리티 순차 패턴을 탐사하는 자료구조로 각 입력 시퀀스에 대한 유틸리티 매트릭스(utility matrix)를 만들고, 탐사되는 각 시퀀스에 대한 입력 시퀀스들의 projected utility matrix를 만들어서 그 시퀀스의 유틸리티 값을 계산한다. 다음 시퀀스들을 확장할 때 포함된 항목들의 SWU를 사용하여 시퀀스들을 만들어가는 USpan 알고리즘을 제안하였다. 두 논문들[12, 13]에서 높은 유틸리티 순차 패턴 마이닝 문제에 대하여 일반적인 정의들이 기술되었다. Alkan 등[14]이 제안한 CRoM은 시퀀스들을 확장해나가는 과정에서 가지치기를 하는 기준으로 설정되었고, 그것을 위한 자료구조는 CSeq를 사용하였다. CSeq는 각 시퀀스 패턴에 대해서 <시퀀스 번호, 시퀀스 내에서 같은 패턴의 순서, 마지막 항목집합 번호, 유틸리티>로 된 튜플들의 집합으로 유틸리티와 그 패턴에서 나머지 유틸리티 값을 계산하는데 이용하였다. Wang 등[15]은 높은 유틸리티 순차 패턴을 탐사하기 위하여 <시퀀스 번호, 트랜잭션 번호, 최대 유틸리티, 나머지 유틸리티>로 구성된 튜플들의 체인으로 된 자료구조를 이용하였다. Zihayat 등[16]도 <시퀀스 번호, 트랜잭션 번호, 유틸리티>로 구성된 튜플들인 시퀀스 유틸리티 리스트를 사용하여 패턴들을 탐사하였다.

3. 높은 유틸리티 순차 패턴 마이닝 개념 및 정의

이 장에서는 높은 유틸리티 순차 패턴 마이닝을 위한 기본 개념과 관련된 정의를 설명한다. 3장의 정의 1~10과 4장의 정의 14~15는 기존 연구에서 사용되는 정의이고[12-16], 4장의 정의 11~13, 16은 본 논문에서 새로 제안하는 정의이다. 집합 $I=\{i_1,i_2,...,i_n\}$을 서로 다른 항목들(items)의 집합이라 하자. 이 집합의 각 항목 $i_k \in I(1 \le k \le n)$ 의 외부 유틸리티(external utility)는 i_k의 단위 이익이나 중요도를 나타내고 $p(i_k)$로 표기하고, 전체 항목들에 대한 것은 이익표(profit table)에 표시한다. Table 1은 Table 2에 주어진 시퀀스 데이터베이스에 대응되는 이익 표를 보여준다. 비어있지 않은 부분집합 $X \subseteq I$는 항목집합이라 부르고 $|X|$는 X에 속한 항목들의 개수를 나타낸다. 간단하게 표시하기 위해 만약 항목집합이 한 개의 항목을 가진다면 중괄호는 생략한다. 한 항목집합에 속한 항목들은 어떤 순서로 나열시킬 수 있기 때문에 보편성을 잃지 않고 항목들은 알파벳 순서로 나열한다고 가정한다. 시퀀스 $s=\langle X_1 X_2 ... X_m \rangle$는 항목집합들의 순서가 있는 리스트이고, $X_j (1 \le j \le m)$은 $X_j \subseteq I$이다.

시퀀스 데이터베이스 S는 튜플들 $\langle sid, s \rangle$의 집합으로 구성되고, sid는 입력 시퀀스를 나타내는 s의 고유한 식별자다. 시퀀스 $s=\langle X_1 X_2 ... X_m \rangle$에서 항목집합 X_j의 각 항목 i는 양의 실수 $q(i,j,s)$와 연관되어 있고, 이 값은 i의 내부 유틸리티(internal utility) 또는 수량이라고 부른다. Table 2에 주어진 예제 시퀀스 데이터베이스에서 s_2에서 두 번째 항목집합 내에 있는 항목 b의 내부 유틸리티(즉, $q(b,2,s_2)$)는 3이다.

정의 1. 두 개의 항목집합 $X_a = \{(i_{a_1},q_{a_1})(i_{a_2},q_{a_2})\cdots(i_{a_n},q_{a_n})\}$와 $Y_b = \{(i_{b_1},q_{b_1})(i_{b_2},q_{b_2})\cdots(i_{b_m},q_{b_m})\}$가 주어지면, 만약 $1 \le k \le n$에 대해서 $i_{a_k}=i_{b_{j_k}} \wedge q_{a_k}=q_{b_{j_k}}$를 만족하는 $1 \le j_1 \le j_2 \le \cdots \le j_n \le m$인 정수들이 존재하는 경우에만 Y_b는 X_a를 포함한다고 하고, $X_a \subseteq Y_b$로 표시한다.

Table 1. Profit Table

item	a	b	c	d	e
external utility	3	4	1	5	2

Table 2. Sequence Database

sid	Sequence
s_1	<(b, 1) {(a, 2) (d, 1)}>
s_2	<{(a, 2) (c, 1)} {(b, 3) (d, 1) (e, 2)}>
s_3	<(d, 2) (b, 2) {(a, 3) (d, 1) (e, 1)}>
s_4	<{(b, 2) (d, 2)} {(a, 2) (e, 3)}>
s_5	<{(a, 1) (b, 1) (d, 2)} {(b, 2) (e, 2)}>

예를 들면, 항목집합 $\{(a,4)(b,1)(e,2)\}$는 $(a,4)$, $\{(a,4)(b,1)\}$, $\{(a,4)(b,1)(e,2)\}$를 포함하지만, 그러나 $\{(a,2)(e,2)\}$와 $\{(a,4)(c,1)\}$을 포함하지 않는다.

정의 2. 시퀀스 $t=\langle X_1 X_2 ... X_k \rangle$가 시퀀스 $s=\langle Y_1 Y_2 ... Y_m \rangle$의 부분시퀀스가 되려면 $X_1 \subseteq Y_{j_1}$, $X_2 \subseteq Y_{j_2}$, ..., $X_k \subseteq Y_{j_k}$를 만족하는 $1 \le j_1 < j_2 < ... < j_k \le m$이 존재하는 조건이 되어야 하고 $t \sqsubseteq s$로 표기한다.

예를 들면, t = <a {b, d}>는 s = <{a, c} {b, d, e}>일 때 $t \sqsubseteq s$가 된다.

정의 3. 만약 시퀀스에 k 항목들이 있는 경우에만 그 시퀀스는 k-시퀀스라 불리고, 그것의 길이는 k이다; 그 시퀀스의 크기는 그 시퀀스에 있는 항목집합들의 개수이다.

예를 들면, Table 2의 시퀀스 s_2의 길이는 항목들이 다섯 개 있어서 5이고, 그 크기는 항목집합들이 두 개 있으므로 2이다.

정의 4. 시퀀스 s에서 j번 째 항목집합의 항목 i의 유틸리티는 $u(i,j,s)=p(i) \times q(i,j,s)$로 정의된다.

예를 들면, $u(b,2,s_2)$ = $p(b) \times q(b,2,s_2)$ = 4×3 = 12.

정의 5. 시퀀스 s에서 j번 째 항목집합에 포함된 항목집합 X의 유틸리티는 다음과 같이 정의된다:

$$u(X,j,s)=\sum_{i \in X} u(i,j,s) \quad (1)$$

예를 들면, $u(\{b,d\},1,s_4)$ = $u(b,1,s_4)$ + $u(d,1,s_4)$ = $4 \times 2 + 5 \times 2$ = 18.

정의 6. t가 s의 서브시퀀스일 때, $1 \le j_1 \le j_2 \le ... \le j_k \le m$이고 $X_1 \subseteq Y_{j_1}$, $X_2 \subseteq Y_{j_2}$, ..., $X_k \subseteq Y_{j_k}$이면, s는 위치 $\langle j_1,j_2,...,j_k \rangle$에서 t의 인스턴스(instance)를 갖는다고 말한다.

예를 들면, t = <a {b, d}>는 s = <{a, c} {b, d, e}>일 때 $t \sqsubseteq s$이고 s는 위치 <1, 2>에서 t의 인스턴스를 갖는다. 한 시퀀스에 여러 개의 인스턴스들이 있는 경우를 상정할 경우에 최대 유틸리티를 계산할 때 이 정의가 적용된다.

정의 7. 시퀀스 t가 시퀀스 s의 서브시퀀스이고 시퀀스 s가 위치 $\langle j_1,j_2,...,j_k \rangle$에서 t의 인스턴스를 가지면, 위치

$\langle j_1,j_2,...,j_k\rangle$에서 t의 인스턴스의 유틸리티는 다음과 같이 정의된다:

$$u(t,\langle j_1,j_2,...,j_k\rangle,s) = \sum_{i=1}^{k} u(X_i,j_i,s) \quad (2)$$

예를 들면, $u(<a\{b,d\}>,<1,2>,s_2) = u(a,1,s_2) + u(\{b,d\},2,s_2) = 9 + (8 + 5) = 22$.

정의 8. 시퀀스 s에서 시퀀스 t의 유틸리티는 $u(t,s)$로 표기되고 s에 속한 t의 모든 인스턴스들의 유틸리티들의 최댓값으로 정의된다:

$$u(t,s) = \max\{u(t,\langle j_1,j_2,...,j_k\rangle,s) | \forall \langle j_1,j_2,...,j_k\rangle : t \sqsubseteq \langle Y_{j_1}, Y_{j_2}, ..., Y_{j_k}\rangle\} \quad (3)$$

예를 들면, $u(<d>,s_3) = \max\{u(<d>,<1>,s_3), u(<d>,<2>,s_3\} = \max\{10, 5\} = 10$.

정의 9. 시퀀스 데이터베이스 S에서 시퀀스 t의 유틸리티는 $u(t)$로 표기되고, 다음과 같이 정의된다:

$$u(t) = \sum_{s\in S \wedge t \sqsubseteq s} u(t,s) \quad (4)$$

예를 들면, $u(\langle a\{b,e\}\rangle) = u(\langle a\{b,e\}\rangle,s_2) + u(\langle a\{b,e\}\rangle,s_5) = 22 + 15 = 37$.

정의 10. 사용자-지정 최소 유틸리티 임계값이 ξ고 시퀀스 데이터베이스 S에서 시퀀스 t에 대해 만약 $u(t) \geq \xi$이면 시퀀스 t는 S에서 높은 유틸리티 순차 패턴(high utility sequential pattern)이라 부른다. 따라서 유틸리티 시퀀스 데이터베이스 S가 주어지고 최소 유틸리티 임계값 ξ가 지정되면, 높은 유틸리티 순차 패턴을 탐사하는 문제는 S에서 ξ를 만족하는 모든 높은 유틸리티 시퀀스들을 추출하는 것이다.

4. HSUL-Span 알고리즘

4.1 시퀀스 유틸리티 매트릭스의 구조

시퀀스 데이터베이스에 속한 각 시퀀스에 대한 유틸리티 매트릭스를 만든다[13, 14]. Table 3은 Table 2의 세 번째 시퀀스에 대한 시퀀스 유틸리티 매트릭스이다. Table 3에서 유틸리티 매트릭스의 식별자는 sid로 표시한다. Table 3의 각 원소는 (u, ru)의 값을 가지는 데, u는 그 항목의 유틸리티를 표시하고, ru는 그 시퀀스에서 그 항목 이후의 나머지-유틸리티(remaining-utility)를 나타낸다. Table 2의 s_3 시퀀스의 첫 번째 항목집합 (d, 2)는 항목 d를 2개를 구매했음을 나타내고, Table 1에서 d는 5의 가중치를 가지므로 이 항목의 유틸리티 u는 10이 된다. 나머지-유틸리티 ru에 대한 정의와 예제를 설명한다.

Table 3. Sequence Utility Matrix of s_3

			sid = 3
items	itemset 1	itemset 2	itemset 3
a	(0, 34)	(0, 24)	(9, 7)
b	(0, 34)	(8, 16)	(0, 7)
d	(10, 24)	(0, 16)	(5, 2)
e	(0, 24)	(0, 16)	(2, 0)

정의 11. 시퀀스 s에서 j번 째 항목집합의 항목 i에서 나머지-유틸리티를 $ru(i,j,s)$로 표기하고, $ru(i,j,s)$는 그 시퀀스에서 그 항목 이후에 나오는 모든 항목들의 유틸리티들의 합으로 정의된다.

예를 들면, $ru(d,1,s_3) = u(b,2,s_3) + u(a,3,s_3) + u(d,3,s_3) + u(e,3,s_3) = 8 + 9 + 5 + 2 = 24$.

4.2 시퀀스 유틸리티 리스트의 구조

각 항목 또는 한 시퀀스의 유틸리티를 계산하거나 항목을 접합하여 패턴들을 확장해가는 과정에서 필요한 정보를 보관한 것이 시퀀스 유틸리티 리스트(sequence utility list)이다. 한 시퀀스 유틸리티 리스트는 그 시퀀스를 포함하는 시퀀스 유틸리티 매트릭스와 그 매트릭스에서 해당 시퀀스의 위치와 그것의 유틸리티 값을 저장한다. 본 논문에서 제안하는 시퀀스 유틸리티 리스트의 각 튜플은 <sid, r, c, u, ru>로 구성된다. 여기서 sid는 해당 항목이나 시퀀스를 가진 시퀀스 유틸리티 매트릭스를 나타내는 고유 식별자를 나타내고 또는 간단히 s로 표시하기도 한다. 본 논문에서 시퀀스 유틸리티 리스트의 튜플은 주로 <s, r, c, u, ru>로 표시한다. 튜플의 나머지 네 값들은 그 매트릭스의 행(row, r)과 열(column, c)을 표시하고 그 곳에 저장된 유틸리티 값(u)과 나머지-유틸리티 값(ru)을 표시한다. 이 유틸리티 리스트에 속한 튜플들은 sid를 기준으로 오름차순으로 정렬되어 있다.

그 예제로 Table 4는 Table 1과 2에서의 항목 b에 대한 항목 유틸리티 리스트를 설명하고 있다. 항목 b를 포함하고 있는 시퀀스의 정보를 포함하고 있는 튜플(tuple)들로 이루어지고 있다. Table 4의 세 번째 튜플은 항목 b가 s_3 시퀀스 유틸리티 매트릭스인 Table 3의 두 번째 행(r = 2)과 두 번째 항목집합(c = 2)에 있고 그 유틸리티 값은 8이고 나머지-유틸리티 값은 16임을 설명하고 있다. Table 4에서 5번

Table 4. Item Utility List of item b

sid	r	c	u	ru
s_1	2	1	4	11
s_2	2	2	12	9
s_3	2	2	8	16
s_4	2	1	8	22
s_5	2	1	4	22
s_5	2	2	8	4

째 튜플과 6번 째 튜플에서는 같은 s_5 시퀀스 유틸리티 매트릭스를 나타내고, 이것은 Table 2의 시퀀스 데이터베이스의 다섯 번째 시퀀스인 s_5에서 항목 b가 첫 번째 항목집합과 두 번째 항목집합에 있는 것을 표시하고, r의 값이 2인 것은 그 시퀀스에서 표시된 4개의 항목들 중에서 2번째 항목이 b임을 의미한다.

정의 12. 시퀀스 t에 대한 유틸리티 리스트 L_t가 주어지면, 그 유틸리티-리스트의 전체 유틸리티(whole utility, wu)는 $wu(t)$로 표기하고 다음과 같이 정의된다:

$$wu(t) = \sum_{s \in L_t} \max\{u \mid \forall u : tuple < s,r,c,u,ru >\} \quad (5)$$

여기서 $s \in L_t$는 L_t에 속해있는 시퀀스 s를 나타내고 같은 시퀀스 번호를 갖는 튜플들은 유틸리티 값들 중에 최댓값으로 더해지고, 그렇지 않으면 단일 유틸리티 값이 더해진다.

예를 들면, Table 4의 항목 b에 대한 항목 유틸리티-리스트에서 전체 유틸리티는 Equation (5)에 의해서 $wu(< b >)$ = 4 + 12 + 8 + 8 + max(4, 8) = 40이 된다. Equation (4)와 비교하면, 결과적으로 $wu(t) = u(t)$가 된다.

정의 13. 시퀀스 t에 대한 유틸리티 리스트 L_t가 주어지면, 그 유틸리티-리스트의 전체 나머지-유틸리티(whole remaining-utility, wru)는 $wru(t)$로 표기하고 다음과 같이 정의된다:

$$wru(t) = \sum_{s \in L_t} \max\{ru \mid \forall ru : tuple < s,r,c,u,ru >\} \quad (6)$$

여기서 같은 시퀀스에서는 제일 먼저 나오는 값이 최댓값이 된다.

예를 들어, Table 4의 s_5를 갖는 튜플들은 다섯 번째와 여섯 번째인데 나머지-유틸리티 값의 특성상 c가 작을수록 그 ru의 값이 크게 된다. Table 4의 항목 b의 $wru(< b >)$ = 11 + 9 + 16 + 22 + max(22, 4) = 80이 된다.

4.3 사전적 순서 트리

높은 유틸리티 순차 패턴(HUSP)을 찾는 다른 알고리즘들[13-16]와 같이 HUSP를 탐사하는 문제의 탐색 공간을 사전적 순서 트리(lexicographic tree)로 나타낼 수 있다. 사전적 순서 트리는 "<>"로 표시되는 루트와 루트를 제외한 노드들로 구성되고, 각 노드는 <t, utility-list>를 가지는데 t는 시퀀스에 해당되는 패턴 스트링이고 utility-list는 Table 4에서 설명한 것과 같이 그 패턴을 이루는 유틸리티-매트릭스의 정보를 보유한 튜플들로 만들어진다.

항목들의 개수가 l개인 l-시퀀스 t가 있다고 하면, $(l+1)$-시퀀스를 형성하기 위해 t의 마지막에 새로운 항목을 추가하는 연산을 접합(concatenation)이라 부른다[13-16]. 접합에는 두 가지 종류가 있는데 t의 크기(항목집합들의 개수)가 늘어나지 않고 길이만 커지는 I-접합과 항목집합들의 개수가 1 증가하는 S-접합이 있다.

정의 14. (I-접합, S-접합) l-시퀀스 t가 주어지면, 만약 $(l+1)$-시퀀스 t'이 t의 마지막에 단일 항목으로 구성된 새로운 항목집합을 추가하여 생성되면 t'은 S-접합 시퀀스라 부른다. 만약 t'이 t의 마지막 항목집합에 한 항목을 삽입하여 생성되면, t'은 I-접합이라 부른다.

예를 들면, <bd>의 I-접합은 <b{de}>이고 S-접합은 <bda>가 된다.

예제 시퀀스 데이터베이스에 대한 사전적 순서 트리의 일부를 Fig. 1에서 보여주고 있다. 조건을 만족하는 항목을 하나씩 접합해 가면서 한 시퀀스를 만들고, 그 시퀀스의 유틸리티가 사용자-지정 최소 유틸리티 임계값보다 크거나 같으면 HUSP가 된다. 그렇지만 후보 패턴으로 만들어지는 시퀀스들의 개수가 너무 많이 생성되지 않도록 하는 것이 중요하다. 유망하지 않은 항목들을 배제하기 위하여 한 시퀀스의 SWU 값을 적용하고 있다[13-16]. 다음은 SWU를 정의하고, 이를 바탕으로 포함 관계인 두 시퀀스들 사이의 SWU 값들의 관계를 정리로 기술한다.

정의 15. 시퀀스 데이터베이스 S에서 한 시퀀스 t의 sequence-weighted utility (SWU)는 $SWU(t)$로 표기되고 다음과 같이 정의된다:

$$SWU(t) = \sum_{t \sqsubseteq s \wedge s \in S} u(s,s) \quad (7)$$

예를 들면, Table 2의 시퀀스 데이터베이스에서 $SWU(\langle c \rangle)$는 시퀀스 <c>가 두 번째 시퀀스인 s_2에만 있으므로, $SWU(\langle c \rangle) = u(s_2,s_2) = 28$이 되고, $SWU(\langle e \rangle)$는 시퀀스 <e>가 포함되어 있는 시퀀스들은 $\{s_2, s_3, s_4, s_5\}$이므로 $SWU(\langle e \rangle) = u(s_2,s_2) + u(s_3,s_3) + u(s_4,s_4) + u(s_5,s_5) = 28 + 34 + 30 + 29 = 121$이 된다.

정리 1. (Sequence-Weighted Downward Closure Property [11, 13, 15]) 유틸리티-기반의 시퀀스 데이터베이스 S, t_2가 t_1을 포함하는($t_1 \sqsubseteq t_2$) 두 시퀀스 t_1과 t_2가 주어지면,

$$SWU(t_2) \leq SWU(t_1) \quad (8)$$

정리 1에 의해서, 어떤 항목이나 시퀀스 t의 $SWU(t) < \xi$이면 안전하게 그 항목이나 시퀀스를 가지치기하여 그 자손들의 생성을 막을 수 있다.

다음 정의는 시퀀스 s에서 시퀀스 t가 항목을 접합해나갈 때 예상되는 최대 유틸리티 값을 계산하는 것이다.

정의 16. 시퀀스 데이터베이스 S에서 시퀀스 t가 항목을 접합해나갈 때 예상되는 최대 유틸리티(sequence expected utility)는 $SEU(t)$로 표기한다. 그러면 S에서 $SEU(t)$를 계산하는 것은 시퀀스 t의 유틸리티 리스트에서 $SEU(t)$를 계산하는 것과 같고 다음과 같이 정의된다:

$$SEU(t) = wu(t) + wru(t) \quad (9)$$

예를 들면, 시퀀스 $<b>$의 $SEU(<b>) = wu(<b>) + wru(<b>) = 40 + 80 = 120$이 된다.

정리 2. 시퀀스 t와 S가 주어지면, t와 t의 각 자손 노드인 t'의 최대 유틸리티 $u(t')$는 $u(t') \leq SEU(t)$.

증명: 시퀀스 t를 갖는 $s \in S$인 각 시퀀스의 정보는 t의 유틸리티 리스트에서 튜플 $<s, r, c, u, ru>$에 그 정보를 저장하고 있다. 튜플의 유틸리티 매트릭스에 관한 정보를 가진 시퀀스 s에서 시퀀스 t를 포함하고 있는 부분의 유틸리티는 u이고 나머지 항목들에서 얻을 수 있는 나머지-유틸리티는 ru의 값을 갖는다. t의 자손 노드인 t'은 항목들을 접합해나가면서 얻을 수 있는 최대한의 유틸리티 값은 그 튜플에서 $u+ru$이다. 즉, 어떤 항목을 접합하더라도 이 값보다 더 큰 유틸리티 값을 얻을 수 없다. 그런데 시퀀스 t의 유틸리티 리스트에서는 한 시퀀스 s에 속한 여러 개의 튜플들이 있을 수 있지만, 정의 12와 13에서 그 시퀀스 s에서 얻을 수 있는 최댓값으로 $wu(t)$와 $wru(t)$가 계산된다. 결과적으로 시퀀스 t를 포함하는 모든 시퀀스들에서 항목들을 접합하면서 만들어지는 t'의 $u(t')$이 얻을 수 있는 최대의 유틸리티 값은 $SEU(t)$가 된다.

정리 2에 의하면, $SEU(t)$는 시퀀스 t의 어떤 자손 시퀀스들의 유틸리티의 상한을 나타낸다. 그러므로 $SEU(t) < \xi$인 시퀀스 t의 모든 자손 노드들은 유망하지 않은 패턴이기 때문에 탐사의 결과에 영향을 주지 않고 안전하게 가지치기를 할 수 있다.

Fig. 1의 예제 트리는 Table 2의 예제 시퀀스 데이터베이스에서 만들어졌고 사용자-지정 최소 유틸리티 임계값 ξ를 30으로 설정하여 설명한다. Fig. 1에서 노드가 되려면 정리 1과 2에 의해서 그 노드의 시퀀스 t가 $SWU(t) \geq \xi$이고, 또한 그 노드의 시퀀스 t에 대한 유틸리티 리스트의 $wu(t)$와 $wru(t)$를 합한 값인 $SEU(t) \geq \xi$ 이어야 이 트리의 노드가 될 수 있다. 예제 시퀀스 데이터베이스에서 항목 <e>의 $SWU(<e>) = 121$이지만, 그 노드의 전체 유틸리티 $wu(<e>) = 16$이고 전체 나머지-유틸리티 $wru(<e>) = 0$이 되어 $SEU(<e>) = 16 < \xi$이기 때문에 이 트리의 노드는 될 수 없다. Table 2의 시퀀스들에서 항목 <a>, <b>, <d>는 이 트리의 노드가 될 수 있는데, 각 노드의 전체 유틸리티 값과 나머지-유틸리티 값이 사용자-지정 최소 유틸리티 임계값보다 크거나 같은 값을 가진다. 항목 <e>는 정리 1과 정의 14에 의해서 I-접합이나 S-접합에 적용되는 항목이기 때문에 이 항목의 유틸리티 리스트는 저장된다. 예제 시퀀스 데이터베이스에서 항목 <a>, <b>, <d>, <e>는 I-접합이나 S-접합에 사용될 수 있는 항목들이다.

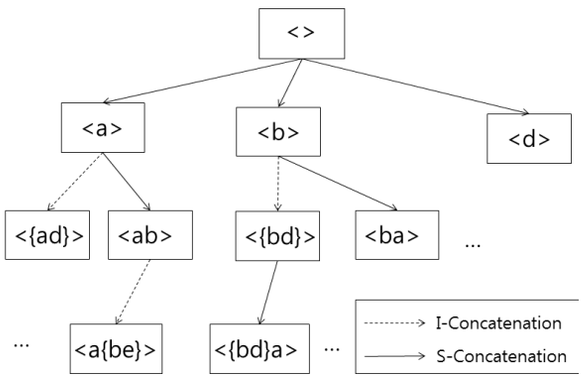


Fig. 1. Lexicographic Tree for the Example in Table 2

4.4 HSUL-Span 알고리즘

높은 유틸리티 순차 패턴(HUSP)들을 탐사하기 위해 제안된 알고리즘은 Fig. 2의 알고리즘 1에서 시작된다. 시퀀스 데이터베이스에서 시퀀스 유틸리티 매트릭스와 항목 유틸리티 리스트를 만들어서 재귀 함수를 부르는 과정은 Fig. 2에서 설명하고, 재귀적으로 패턴을 성장하면서 HUSP를 찾는 과정은 Fig. 5에서 설명한다. 제안된 알고리즘의 이름은 패턴을 탐사하는 과정에서 생성되는 시퀀스 유틸리티 리스트들 중에서 높은 유틸리티를 찾는 과정이므로 **H**igh **S**equence **U**tility **L**ist - **Span**ning (HSUL-Span)으로 명명하였다.

Fig. 2에서 메인 알고리즘을 java-like 언어로 기술한다. 줄 3에서 입력 시퀀스 데이터베이스를 읽어서 각 항목의

Algorithm 1. HSUL-Span-Main(S, ξ)

Input: The sequence database S and the minimum utility threshold ξ
Output: All high utility sequential patterns

```
1:  Map<Integer, Integer> mapItemToSWU = new
                                HashMap<Integer, Integer>();
2:  List<TrimSequence> SDB_trim = new
                                ArrayList<TrimSequence>();
3:  SDB_trim = ScanAndTrimSDB(S, ξ, mapItemToSWU,
                                SDB_trim);
4:  List<UtilityMatrix> MAT_seq = new
                        ArrayList<UtilityMatrix>(|SDB_trim| );
5:  Map<Integer, ItemNode> IUL−SET = new
                                hashMap<Integer, ItemNode>();
6:  IUL−SET = SeqMatrixAndItemUtilList(SDB_trim, ξ,
                                mapItemToSWU, MAT_seq);
7:  for each item i ∈ IUL−SET with SWU(i) ≥ ξ
8:      if SEU(i) ≥ ξ then      // refer to Def. 16 and Eq. (9)
9:          if wu(i) ≥ ξ then   // refer to Def. 12 and Eq. (5)
10:             output i;
11:         HSUL-Span(ξ, i, UtilityList(i));
```

Fig. 2. HSUL-Span-Main Algorithm

Algorithm 2. ScanAndTrimSDB(S, ξ, mapItemToSWU, SDB_{trim})

```
Input: The sequence database S and the minimum utility
       threshold ξ
Output: Trimmed sequence database, SDB_trim, and
        mapItemToSWU with SWU(i)
1:  Map<Integer, Integer> mapItemToSWU2 = new
                              HashMap<Integer, Integer>();
2:  totalUtility = 0;
2:  for each sequence s_i ∈ S with seqUtil   // scan the S
4:     totalUtility += seqUtil;      // add this sequence utility
5:     for each item j ∈ s_i
6:        swu = mapItemToSWU2.get(j);
7:        swu = (swu == NULL) ? seqUtil : swu + seqUtil;
8:        mapItemToSWU2.put(j, swu);
9:  ξ = totalUtiltiy × ξ;
10: SDB_trim = S;
11: while (true) {
12:    for each sequence s_i ∈ SDB_trim with seqUtil   // trim
13:       swu_i = 0;
14:       for each item j ∈ s_i with mapItemToSWU2.get(j) ≥ ξ
15:          swu_i += j_util;          // item-utiltiy
16:          save item with item-utility in (j, j_util);
17:       TrimSequence seqT = new
                    TrimSequence(listOf(j, j_util) with swu_i);
18:       SDB_trim.add(seqT); // add the trimmed sequence
19:    mapItemToSWU.clear();
20:    for each sequence s_i ∈ SDB_trim with seqUtil
21:       for each item j ∈ s_i
22:          swu = mapItemToSWU.get(j);
23:          swu = (swu == NULL) ? seqUtil : swu + seqUtil;
24:          mapItemToSWU.put(j, swu);
25:    if (mapItemToSWU.size() == mapItemToSWU2.size())
26:       return SDB_trim;
27:    mapItemToSWU2 = mapItemToSWU;  // in SDB_trim
28: }
```

Fig. 3. ScanAndTrimSDB Algorithm

SWU 값이 최소 유틸리티 값보다 크거나 같은 항목만 남기는 함수를 부른다. 이 함수는 Fig. 3의 알고리즘 2에서 상세히 설명한다. 알고리즘 1의 줄 6에서 남아있는 항목들로 이루어진 각 시퀀스에 대하여 시퀀스 유틸리티 매트릭스와 각 항목의 유틸리티 리스트를 만드는 함수를 부른다. 이 함수는 Fig. 4의 알고리즘 3에서 서술된다. 줄 7~11에서는 한 항목의 유틸리티 리스트에서 계산된 SEU의 값이 ξ보다 크거나 같으면 패턴 트리의 첫 번째 레벨의 노드가 되어서 항목의 자손 시퀀스들에 대한 HUSP를 탐사하는 함수를 부른다. 줄 10에서 그 항목의 전체 유틸리티 값이 ξ보다 크거나 같으면 HUSP가 출력된다.

Fig. 3의 알고리즘 2는 입력인 시퀀스 데이터베이스를 읽어서 각 항목의 SWU를 구하고 사용자 -지정 최소 유틸리티 임계값을 계산한다. 알고리즘 2의 줄 10~28은 각 시퀀스에서 항목의 SWU 값이 사용자-지정 최소 유틸리티 임계값 보다 같거나 큰 항목만 그 시퀀스에 남긴다. 남겨진 항목들의 유틸리티 값을 더하면 그 시퀀스의 새로운 시퀀스

Algorithm 3. SeqMatrixAndItemUtilList(SDB_{trim}, ξ, mapItemToSWU, MAT_{seq})

```
Input: The trimmed sequence database SDB_trim, the minimum
       utility threshold ξ, and the mapping item to SWU,
Output: The sequence utility matrix, MAT_seq, and
        the set of item utility lists, IUL-SET
1:  List<UtilityMatrix> MAT_seq = new
                    ArrayList<UtilityMatrix>(|SDB_trim| );
    // make the utility matrix
2:  for each sequence s_i ∈ SDB_trim with seqUtil
       // refer to Def. 11 and Table 3
3:     UtlityMatrix matrix = new UtilityMatrix(s_i);
4:     MAT_seq.add(matrix);
6:  for each item i ∈ mapItemToSWU
       // make the UtilityList(i), refer to Table 4
7:     if (SWU(i) ≥ ξ) then
8:        ArrayList<ItemUtilityList> itemUtilLists = new
                                 ArrayList<ItemUtilityList>();
9:        for each matrix ∈ MAT_seq
10:          itemUtilList.add(new ItemUtilityList(matrix,
                                 row, itemset, util, remUtil));
11:       ItemNode i_node = new ItemNode(i, itemUtilLists,
                                       sumUtil, sumRemUtil);
12:       IUL-SET.put(i, i_node);
13: return IUL-SET;
```

Fig. 4. SeqMatrixAndItemUtilList Algorithm

유틸리티가 되고, 이 값은 이전 시퀀스 유틸리티보다 작거나 같게 된다. 더 이상의 항목들의 감소가 일어나지 않으면, 줄 26에서 남겨진 시퀀스 데이터베이스를 반환한다.

Fig. 4의 알고리즘 3은 남아있는 시퀀스 데이터베이스의 각 시퀀스에 대한 유틸리티 매트릭스를 Table 3과 같이 만든다. 알고리즘 3의 줄 6~12 에서는 남겨진 시퀀스 데이터베이스에서 각 항목의 SWU의 값이 ξ보다 크거나 같은 항목에 대하여 Table 4와 같이 항목 유틸리티 리스트를 만든다. 이 때 만들어진 항목 유틸리티 리스트들은 I-접합이나 S-접합의 후보 원소가 된다.

알고리즘 4의 줄 1에서 I-접합을 위하여 필요한 항목들인 I-Set을 만들고, 줄 2에서 그 집합에 포함된 항목들 중에서 한 항목의 SEU가 ξ보다 작은 것은 유망하지 않은 항목이므로 그 항목은 제외된다. 줄 3에서 I-Set에 남아있는 각 항목을 줄 4의 I-접합 함수를 사용하여 새로운 시퀀스 패턴인 t'의 유틸리티 리스트를 만든다. 줄 4에서 만들어진 시퀀스 t'는 HUSP의 후보(candidate HUSP)가 된다. 정의 16에 의한 t'의 시퀀스 예상 유틸리티인 $SEU(t')$을 계산하여 그 값이 ξ보다 크거나 같으면 t'은 잠재적인 높은 유틸리티 순차 패턴(potential HUSP)이 된다. 줄 6에서 t'의 유틸리티인 $u(t')$이 ξ 보다 크거나 같으면 실제 HUSP가 된다. 그리고 t'은 다음 순차 패턴을 탐사하기 위해 재귀 호출된다. 알고리즘 4의 줄 9~16도 S-접합을 통한 새로운 시퀀스를 만들고 후보 HUSP, 잠재적인 HUSP, 그리고 실제 HUSP를 찾는 과정을 보여준다.

Algorithm 4. HSUL-Span(ξ, t, $UtilityList(t)$)

Input: The minimum utility threshold ξ and a sequence t with utility-list $UtilityList(t)$.
Output: All high utility sequential patterns
1: $I-Set$ = find$I-Set$($UtilityList(t)$);
2: delete low SEU items from $I-Set$
3: **for each** item $i \in I-Set$ **do** // candidate HUSP
4: $UtilityList(t')$ = I-Concatenation($UtilityList(t)$, $UtilityList(i)$);
5: **if** $SEU(t') \geq \xi$ **then** // potential HUSP
6: **if** $u(t') \geq \xi$ **then** // HUSP
7: output t';
8: HSUL-Span(ξ, t', $UtilityList(t')$);
9: $S-Set$ = find$S-Set$($UtilityList(t)$);
10: delete low SEU items from $S-Set$;
11: **for each** item $i \in S-Set$ **do**
12: $UtilityList(t')$ = S-Concatenation($UtilityList(t)$, $UtilityList(i)$);
13: **if** $SEU(t') \geq \xi$ **then**
14: **if** $u(t') \geq \xi$ **then**
15: output t';
16: HSUL-Span(ξ, t', $UtilityList(t')$);

Fig. 5. HSUL-Span Algorithm

알고리즘 4의 줄 1에서 I-Set을 UtililtyList(t)에서 구하는 알고리즘 5가 Fig. 6에서 기술된다. 알고리즘 5의 줄 2에서 각 튜플은 <s, r, c, u, ru>로 표기된다. 튜플에서 s가 가리키는 유틸리티 매트릭스의 현재 항목집합(c)에서 현재 항목의 행(r) 다음에 나오는 항목들 중에서 유틸리티 값이 존재하면 그 항목이 I-Set의 원소가 된다. 알고리즘 5의 줄 7에서 현재 튜플에서 항목인 item을 접합하면 확장되는 시퀀스의 예상 유틸리티(SEU)의 값은 현재 튜플의 유틸리티 u, 그리고 Table 3에서와 같이 그 항목의 해당 셀 ($s.utilityMatrix[row][c]$)에 있는 값인 유틸리티 u와 나머지-유틸리티 ru를 더한 값이 된다. 그 항목이 처음 나타나면 그 값을 줄 9~12에서와 같이 저장하고, 그렇지 않고 그 항목으로 저장된 마지막 시퀀스 유틸리티 매트릭스의 sid가 현재 sid와 다르면 줄 14~15에서 기존의 SEU의 값에 그 값을 추가해서 더해진다.

알고리즘 4의 줄 4에서 수행하는 I-접합에 대한 상세 과정이 Fig. 7의 알고리즘 6에서 기술한다. 알고리즘 6에서 UtililtyList(t)에 속한 각 튜플 T_t = <s_t, r_t, c_t, u_t, ru_t>에 대하여 만약 $s_t = s_{ij}$과 $c_t = c_{ij}$을 만족하는 UtilityList(i)에 속한 j-번째 튜플 T_{ij} = <s_{ij}, r_{ij}, c_{ij}, u_{ij}, ru_{ij}>가 있으면 I-접합의 결과인 새 튜플 $T_{t'}$ = <s_{ij}, r_{ij}, c_{ij}, $u_t + u_{ij}$, ru_{ij}>를 UtilityList(t')에 추가한다. 알고리즘 6의 줄 4에서 UtilityList(i)에서 $s_t.sid$를 찾는 이진검색 BinarySearchB()은 $s_t.sid$를 찾으면 같은 sid가 있는 인덱스들 중에서 제일 작은 인덱스를 반환한다. 그렇지 않으면, 그 값보다 큰 다음 값에 해당하는 인덱스에 +1을 한 후에 그 값의 마이너스 값을 반환한다. 마이너스 값의 인덱스는 알고리즘 6의 줄 17에서 해당 인덱스로 변환하는데 -1을 하는 것은 인덱스가

Algorithm 5. find$I-Set$($UtilityList(t)$)

Input: The Sequence Utility List of t, $UtilityList(t)$.
Output: $I-Set$ for I-Concatenation items of the sequence t.
1: Map<Integer, Pair> mapItemToSEU = **new** HashMap<Integer, Pair>();
2: **for each** tuple T = <s, r, c, u, ru> $\in$ $UtilityList(t)$ {
3: **for** (row = $r+1$; row < s.itemNames().$length$; row++) {
4: item = s.itemNames[row];
5: **if** ($s.utilityMatrix[row][c].u$ > 0) {
6: Pair curSEU = mapItemToSEU.get(item);
7: tempSEU = u + $s.utilityMatrix[row][c].u$ + $s.utilityMatrix[row][c].ru$;
8: **if** (curSEU == null) {
9: Pair pair = **new** Pair(); // (lastSID, seu)
10: pair.lastSID = $s.sid$;
11: pair.seu = tempSEU;
12: mapItemToSEU.put(item, pair);
13: } **else if** (curSEU.lastSID != $s.sid$) {
14: curSEU.lastSID = $s.sid$;
15: curSEU.seu += tempSEU;
16: }
17: }
18: }
19: }
20: **return** $I-Set$ from mapItemToSEU;

Fig. 6. find$I-Set$ Algorithm

Algorithm 6. I-Concatenation($UtilityList(t)$, $UtilityList(i)$)

Input: The Sequence Utility Lists of t and i, $UtilityList(t)$ and $UtilityList(i)$
Output: $UtilityList(t')$ with sequence $t' = t \cdot i$
1: ArrayList<UtilityList> tPrimeUtilityList = **new** ArrayList<UtilityList>();
2: j = 0; // start index for binary search in $UtilityList(i)$
3: **for each** tuple T_t = <s_t, r_t, c_t, u_t, ru_t>$\in$ $UtilityList(t)$ {
4: j = BinarySearchB($UtilityList(i)$, $s_t.sid$, j, $UtilityList(i).size()$);
5: **if** (j >= 0) {
// j-th tuple of $UtilityList(i)$
6: tuple T_{ij} = <s_{ij}, r_{ij}, c_{ij}, u_{ij}, ru_{ij}>;
7: start_j = j;
8: **do** { // for the same utility matrix
9: **if** (c_t == c_{ij})
10: tPrimeUtilityList.add(new UtilityList(s_{ij}, r_{ij}, c_{ij}, $u_t + u_{ij}$, ru_{ij}));
11: j++; // index for the next tuple
12: **if** (j >= $UtilityList(i).size()$) **break**;
13: **else** tuple T_{ij} = <s_{ij}, r_{ij}, c_{ij}, u_{ij}, ru_{ij}>;
14: } **while** (s_t == s_{ij});
// tuples of $UtilityList(t)$ may have the same sid
15: j = start_j;
16: } **else** {
17: j = -j - 1; // start index of BinarySearchB()
18: **if** (j >= $UtilityList(i).size()$) **break**;
19: }
20: }
21: **return** tPrimeUtilityList with sequence $t' = t \cdot i$;

Fig. 7. I-Concatenation Algorithm

0인 경우에 대한 대처 방법이다. 그러면 이진검색의 검색 범위의 시작 인덱스인 j가 점차적으로 증가하여 검색 범위가 좁혀진다.

알고리즘 4의 줄 9와 12에 있는 함수 findS-*Set*()와 S-Concatenation()는 알고리즘 4의 줄 1과 4에 있는 함수와 비슷해서 상세 과정은 생략하고 구하는 방법을 간략히 기술한다. 줄 9의 함수는 UtililtyList(t)의 각 튜플의 <s_t, r, c, u, ru>에서 s_t가 가리키는 유틸리티 매트릭스에서 현재 항목집합인 c 다음에 나오는 항목집합에 속하는 모든 항목들에 대해 SEU 값을 계산한다. 그리고 줄 12의 S-접합은 UtililtyList(t)에 속한 각 튜플 T_t = <s_t, r_t, c_t, u_t, ru_t>에 대하여 만약 $s_t = s_i$과 $c_t < c_i$을 만족하는 UtilityList(i)에 속한 튜플 T_i = <s_i, r_i, c_i, u_i, ru_i>가 있으면 S-접합의 결과인 새 튜플 $T_{t'}$ = <s_i, r_i, c_i, $u_t + u_i$, ru_i>를 UtilityList(t')에 추가한다.

5. 실험 결과

제안된 알고리즘의 성능을 평가하기 위해 실험은 64GB RAM을 가진 Intel(R) Core(TM) i5 3.30GHz 컴퓨터에서 실행되었다. 제안된 알고리즘과 비교하는 알고리즘들은 JAVA 언어로 구현되었다. 실험에 사용된 데이터는 널리 알려지고 많이 사용되고 있는 IBM data generator[4]로 만들어진 합성 데이터세트 DS1과 DS2, 그리고 SPMF 홈페이지[17]에서 공개된 실제 데이터세트인 kosarak 25k와 retail 데이터세트이다. DS1은 D10C8T4N1S4I2과 DS2는 D100C8T6N10S6I2.5의 형식을 가진다[4, 12-16]. 데이터세트에서 각 항목의 외부 유틸리티는 파라미터 (μ, σ) = (2.0, 0.7)로 설정된 log-normal 분포를 사용하여 1과 100 정도 사이의 값이 생성되도록 하였고, 각 항목의 내부 유틸리티 값은 1과 10 사이의 값이 균일분포로 생성되게 하였다[12, 15, 16]. 실험에 사용된 데이터세트의 특성과 각 파라미터의 의미는 Table 5에서 기술하고 있다.

Table 5. Characteristics of Synthetic and Real Datasets

Dataset	D	C	T	N	Type (domain)
DS1: D10C8T4N1S4I2	10K	8	4	1,000	Synthetic[3]
DS2: D100C8T6N10S6I2.5	100K	8	6	10,000	Synthetic[3]
kosarak 25k	25K	3	5.70	14,804	Real[15]
retail	8,206	3	7.19	8,191	Real[15]

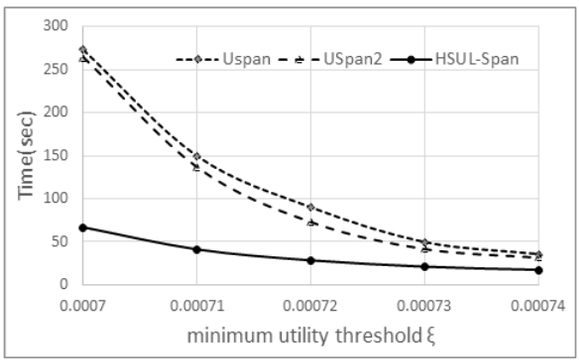
Parameter description

D	Number of sequences
C	Average number of itemsets per sequence
T	Average number of items per itemset
N	Number of distinct items
S	Average number of itemsets in a potential maximal sequential pattern
I	Average number of items in an itemset of a potential maximal sequential pattern

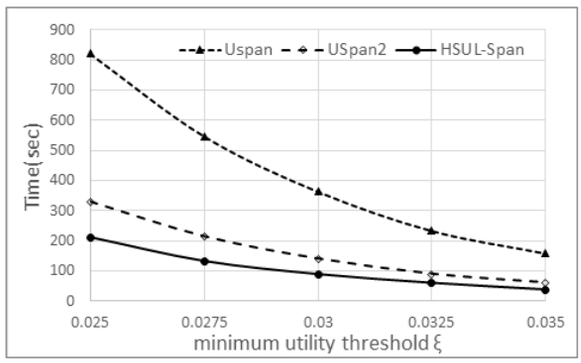
최근의 알고리즘들[14-16]은 성능 평가에서 USpan[13]과 비교하여 실험결과를 서술하였다. 본 논문에서도 제안된 알고리즘과 USpan 알고리즘[13]의 성능 평가를 위해 후보 패턴들의 개수, 실행시간, 그리고 사용된 메모리 사용량을 측정하였다. 홈페이지 SPMF[15]에서 공개되어 있는 USpan 프로그램은 논문 [13]에서 제안된 것과 다르게 구현되어 있기 때문에 본 논문에서는 그것을 USpan2[17]라 부른다. 그 차이점은 알고리즘 4의 줄 1과 9에서와 같이 접합 항목집합 I-Set과 S-Set을 구하는 과정에서 각 항목의 예상 유틸리티를 계산하는데 있다. USpan[13]에서는 자료구조인 projected utility matrix에서 조건에 맞는 각 항목의 예상 유틸리티를 정의 15에 따라 그 항목의 SWU를 더해서 구하고, USpan2 [17]에서는 각 항목의 예상 유틸리티를 그 항목의 유틸리티와 나머지-유틸리티를 더해서 구한다. 제안된 HSUL-Span은 알고리즘 5의 줄 7에서 유틸리티 리스트의 튜플의 유틸리티, 후보 항목의 유틸리티와 나머지-유틸리티 값을 더하여 예상 유틸리티 값을 구한다. 세 개의 알고리즘들에서 만들어지는 I-Set과 S-Set에 속한 항목들로 접합하여 만들어지는 새로운 시퀀스들을 후보 높은 유틸리티 시퀀스 패턴들(candidate HUSP)이라 부른다. 이들의 개수가 알고리즘에 따라 크게 차이가 나는 것을 다음 실험 결과에서 설명한다.

Fig. 8에서는 네 개의 데이터세트에 대하여 세 알고리즘들의 실행시간을 그래프로 비교하고 있다. 그래프에서 사용자-지정 최소 유틸리티 임계값은 ξ로 표기하였다[13, 15]. DS1 데이터세트에서는 제안된 알고리즘인 HSUL-Span이 USpan2에 비해서 46~74% 범위의 성능 개선이 되고, USpan에 비해서는 52~75%의 성능 개선을 보여준다. kosarak 25k와 retail 데이터세트에서는 제안된 알고리즘이 USpan2에 비해 35%와 42% 정도의 성능개선이 실행 시간 측면에서 이루어지고 있다. DS1에 비해 상대적으로 데이터가 희박하게 구성된 DS2 데이터세트에서는 HSUL-Span이 USpan2와 USpan에 비해서 각각 13%와 22% 정도의 성능 개선이 되었다.

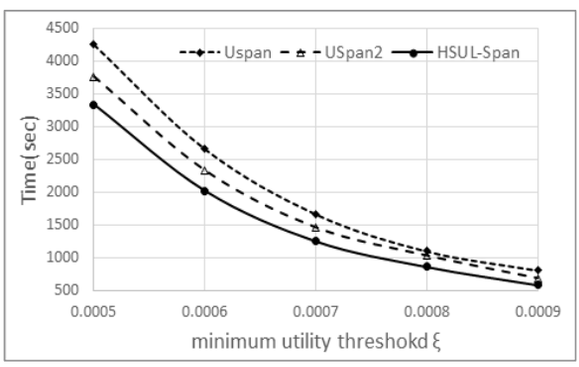
이러한 실행 시간에서 성능 개선은 제안된 알고리즘인 HSUL-Span에서 생성되는 후보 HUSP들이 다른 알고리즘들에 비해 작게 생성되고 시퀀스 유틸리티 리스트에서 HUSP를 찾는 처리 시간이 감소된다. Table 6에서 각 알고리즘별로 각 데이터세트에서 주어진 최소 유틸리티 임계값 ξ의 값에 따라 생성되는 후보 HUSP들의 개수(|CHUSP|), 잠재적인 HUSP들의 개수(|PHUSP|), 최종적으로 찾아진 HUSP들의 개수(|HUSP|)를 보여준다. Table 6에서는 HSUL-Span에 의해 생성되는 CHUSP의 개수가 전반적으로 USPan2에 비해서 50%에서 90%정도까지 줄어드는 것을 보여준다.



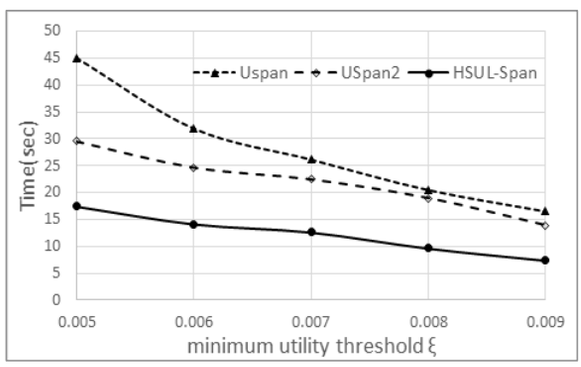
(A) DS1



(B) kosarak 25k

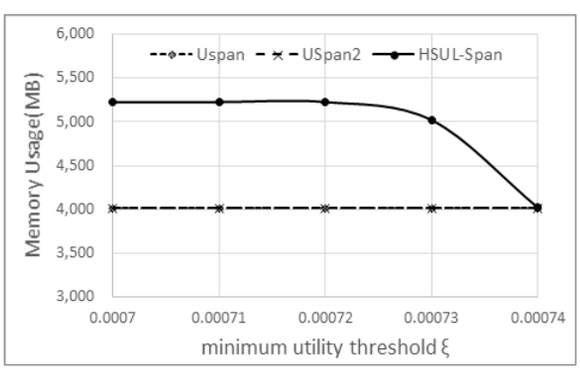


(C) DS2

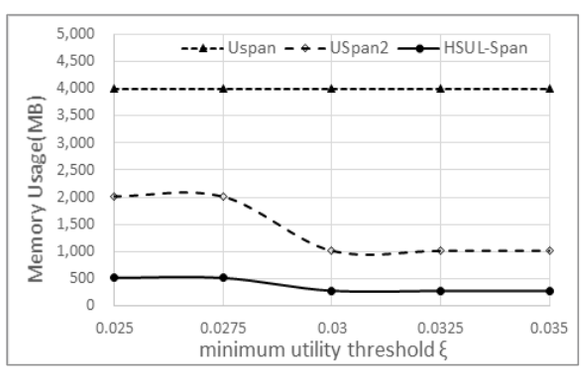


(D) retail

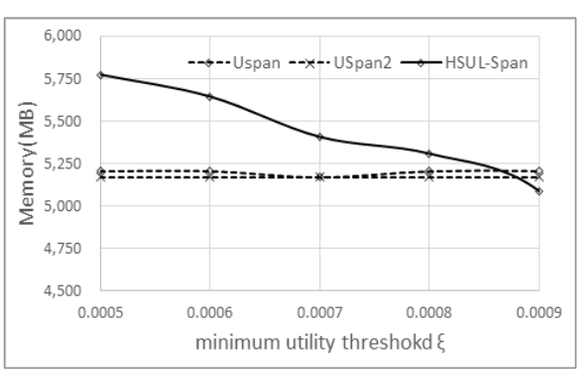
Fig. 8. Execution Time on the Four Datasets



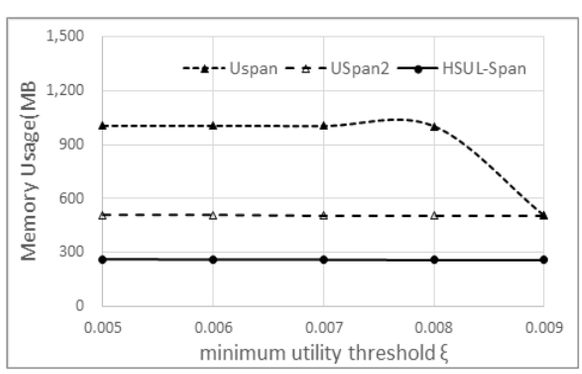
(A) DS1



(B) kosarak 25k



(C) DS2



(D) retail

Fig. 9. Memory Usage on the Four Datasets

Table 6. Number of Candidate HUSPs, Potential HUSPs, and HUSPs on the Four Datasets

dataset	ξ	Algorithm	|CHUSP|	|PHUSP|	|HUSP|
DS1	0.00072	USpan	340,114,202	2,486,462	36,170
		USpan2	332,786,417	2,486,405	36,170
		HSUL-Span	4,989,640	2,476,747	36,170
kosarak 25k	0.03	USpan	5,230,107	270,276	467
		USpan2	866,289	270,276	467
		HSUL-Span	335,312	266,368	467
DS2	0.0007	USpan	41,366,162	5,831,960	1,058
		USpan2	22,981,781	5,831,934	1,058
		HSUL-Span	7,844,119	5,829,623	1,058
retail	0.007	USpan	988,669	98,592	10,105
		USpan2	156,649	98,591	10,105
		HSUL-Span	98,200	98,135	10,105

알고리즘 1의 줄 3에서 부르는 ScanAndTrimSDB 함수에서 트리밍되는 항목들의 개수가 입력 데이터세트에 따라 달라진다. 실험 결과에서 합성 데이터세트인 DS1과 DS2가 입력인 경우에는 항목들을 줄이는 트리밍 효과가 작아서 줄어드는 항목들의 개수는 입력 시퀀스 데이터베이스에 있는 항목들의 10% 이하가 된다. 그러나 실제 데이터세트인 kosarak 25k와 retail에서는 항목들을 줄이는 ScanAndTrimSDB 함수에서 남아있는 항목들의 개수는 입력 시퀀스 데이터베이스에 있는 항목들의 개수보다 아주 작다. 예들 들면, 입력이 kosarak 25k 데이터세트이고 ξ = 0.03인 경우 경우에 ScanAndTrimSDB 함수에서 줄어드는 항목들의 개수는 99% 정도가 된다. 이 경우에 항목 유틸리티 리스트들의 개수가 작아진다. 이것은 패턴 확장 단계인 I-Set나 S-set에서 접합되는 후보 항목들의 개수도 작아지는 것을 의미하기 때문에 전체적인 실행시간을 줄이는 효과를 얻을 수 있다.

Fig. 9는 각 알고리즘이 실행될 때 사용되는 메모리의 사용량을 그래프로 보여준다. HSUL-Span 알고리즘의 메모리 사용량은 다른 두 알고리즘들에 비해 Fig. 9 (B)와 (D)에서는 작고 Fig. 9 (A)와 (C)에서는 30%와 10% 정도 증가되는 것을 보여준다. HSUL-Span에서 항목 또는 시퀀스 유틸리티 매트릭스들과 후보 패턴들의 시퀀스 유틸리티 리스트들을 만드는 것에 메모리가 사용된다. 패턴의 접합에 사용되는 항목 유틸리티 리스트들의 개수가 HSUL-Span의 메모리 사용량에 영향을 준다. 그러므로 ScanAndTrimSDB 함수의 결과에서 남아있는 항목들의 개수가 아주 작은 kosarak 25와 retail인 경우에 HSUL-Span의 메모리 사용량은 다른 두 알고리즘들에 비해 작다. 그렇지만 DS1과 DS2 데이터 세트가 입력일 때 남아있는 항목들이 입력과 거의 비슷한 경우에 HSUL-Span의 메모리 사용량은 커진다. Fig. 9 (C)에서 사용자-지정 최소 유틸리티 임계값이 커짐에 따라 HSUL-Span의 메모리 사용량도 점차적으로 줄어드는 것은 ScanAndTrimSDB 함수에서 남아있는 항목들의 개수가 점차적으로 줄어들기 때문이다.

6. 결 론

본 논문에서는 높은 유틸리티 순차 패턴을 탐사하는 새로운 알고리즘인 HSUL-Span을 제안하였다. 패턴을 확장해 나가는 과정에서 필수적인 정보를 추출해낼 수 있도록 새로운 자료구조를 사용하였다. 이 자료구조는 패턴 또는 시퀀스에 관한 정보를 보관하는 시퀀스 유틸리티 리스트이다. 시퀀스 유틸리티 리스트를 이용하여 패턴에 항목을 접합해 나갈 때 후보 항목들을 찾는 과정과 현재 패턴에 그 후보 항목을 접합하여 새로운 패턴을 만드는 과정을 자세히 기술하였다. 제안된 시퀀스 유틸리티 리스트에서 최대 예상 유틸리티 값인 SEU를 정의하고, 한 시퀀스의 SEU가 사용자-지정 최소 유틸리티 임계값보다 작으면 더 이상의 패턴 확장이 일어나지 못하도록 그 시퀀스를 가지치기하였다. 실험결과에서는 여러 합성 데이터와 실제 데이터를 적용하여 제안된 알고리즘의 성능이 기존의 알고리즘들에 비해 우수함을 보여주었다.

References

[1] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” in *Proceedings of the 20th Very Large Data Base Conference*, Santiago, pp.487-499, 1994.

[2] J. S. Park, M.-S. Chen, and P. S. Yu, “An effective hash-based algorithm for mining association rules,” in P*roceedings of the 1995 ACM SIGMOD international Conference on Management of Data*, San Jose, pp.175-186, 1995.

[3] J. Han, J. Pei, and Y. Yin, “Mining Frequent Patterns without Candidate Generation,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, pp.1-12, 2000.

[4] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, pp.3-14, 1995.

[5] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, “PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth,” in *Proceedings 17th International Conference on Data Engineering*, Heidelberg, pp.215-224, 2001.

[6] N. R. Mabroukeh and C. I. Ezeife, “A Taxonomy of Sequential Pattern Mining Algorithms,” *ACM Computing Surveys,* Vol.43, No.1, Article 3, 2010.

[7] Y. Liu, W. Liao, and A. Choudhary, “A two-phase algorithm for fast discovery of high utility itemsets,” in *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Hanoi, pp.689-695, 2005.

[8] B.-S. Jeong, C. F. Ahmed, I. Lee, and H. Yong, “High utility pattern mining using a prefix-tree,” *Journal of KIISE: Database*, Vol.36, No.5, pp.341-351, 2009.

[9] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol.21, No.12, pp.1708-1721, 2009.

[10] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-Growth: an efficient algorithm for high utility itemset mining," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, pp.253-262, 2010.

[11] S. Lee and J. S. Park, "High Utility Itemset Mining Using Transaction Utility of Itemsets," *KIPS Transactions on Software and Data Engineering,* Vol.4, No.11, pp.499-508, 2015.

[12] C. F. Ahmed, S. K. Tanbeer, and B. Jeong. "A novel approach for mining high-utility sequential patterns in sequence databases," *Electronics and Telecommunications Research Institute Journal*, Vol.32, No.5, pp.676-686, 2010.

[13] J. Yin, Z. Zheng, and L. Cao. "USpan: An efficient algorithm for mining high utility sequential patterns," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Beijing, pp.660-668, 2012.

[14] O. K. Alkan and P. Karagoz, "CRoM and HuspExt: Improving Efficiency of High Utility Sequential Pattern Extraction," *IEEE Transactions on Knowledge and Data Engineering*, Vol.27, No.10, pp.2645-2657, 2015.

[15] J.-Z. Wang, J.-L. Huang, and Y.-C. Chen, "On efficiently mining high utility sequential patterns," *Knowledge and Information Systems*, Vol.49, Issue 2, pp.597-627, 2016.

[16] M. Zihayat, C.-W. Wu , A. An and V. S. Tseng, "Efficiently Mining High Utility Sequential Patterns in Static and Streaming Data," *Intelligent Data Analysis*, Vol.21, No.S1, pp.S103-S135, 2017.

[17] P. Fournier-Viger, An Open-Source Data Mining Library [Internet], http://www.philippe-fournier-viger.com/spmf/index.php, 2017.

박 종 수

http://orcid.org/0000-0003-0892-6812

e-mail : jpark@sungshin.ac.kr

1981년 부산대학교 전기기계공학과(학사)

1983년, 1990년 한국과학기술원 전기 및 전자공학과(석・박사)

1933년~1986년 국방부 군무설계기좌

1994년~1995년 IBM Watson 연구소 객원연구원

1990년~현 재 성신여자대학교 IT학부 교수

관심분야 : Data Mining, Deep Learning, Transportation geography