저작자표시-비영리-변경금지 2.0 대한민국

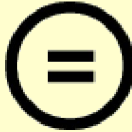
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

박 종 수 교수지도
석사학위 청구논문

유틸리티-리스트 기반의 Top-K 높은 유틸리티 패턴 마이닝

2016

성신여자대학교 대학원
컴퓨터학과
이 세 린

유틸리티-리스트 기반의 Top-K 높은 유틸리티 패턴 마이닝

박 종 수 교수지도

이 논문을 석사학위논문으로 제출함

2015년 11월

성신여자대학교 대학원

컴퓨터학과

이 세 린

인 준 서

이세린의 석사학위 논문으로 인준함

2015년 11월

심사위원장________________(인)

심 사 위 원________________(인)

심 사 위 원________________(인)

성신여자대학교 대학원

논 문 개 요

Top-k 높은 유틸리티 패턴 마이닝은 사용자가 지정한 k 값을 이용해 트랜잭션 데이터베이스에서 항목의 유틸리티를 기준으로 상위 k개의 항목집합을 찾아내는 것이다. 기존의 top-k 높은 유틸리티 패턴 알고리즘은 재귀적으로 트리를 구축하여 패턴을 성장시키는 기법을 기반으로 두 단계에 걸쳐 마이닝을 수행한다. 이는 많은 후보 생성과 그 후보들의 실제 유틸리티를 계산하기 위한 추가적인 데이터베이스 스캔을 필요로 한다. 본 논문에서는 효율적으로 top-k 높은 유틸리티 패턴을 찾기 위해 새로운 알고리즘, TKUL-Miner를 제안한다. 이 알고리즘은 탐색 트리에서 항목집합을 마이닝하기 위해 각 노드에 필요한 정보를 저장하는 유틸리티-리스트 구조를 이용한다. 제안하는 알고리즘은 패턴의 최대 유틸리티가 높은 구간을 먼저 탐색하는 탐색 순서 전략으로 최소 유틸리티를 빠르게 증가시킨다. 그리고 보다 작은 유틸리티 상한 값을 계산하는 두 개의 추가적인 전략들로 유망하지 않은 항목집합들을 효과적으로 가지치기 한다. 다양한 데이터에 대한 실험 결과를 통해 제안하는 알고리즘이 최신 알고리즘들 보다 실행시간과 사용 메모리 관점에서 모두 성능을 개선하고, 특히 조밀한 분포의 데이터인 경우와 평균 트랜잭션의 길이가 긴 경우 제안 알고리즘의 성능이 더욱 효율적이라는 것을 보여준다.

주제어: 높은 유틸리티 항목집합, top-k 패턴 마이닝, 유틸리티-리스트 구조, 데이터 마이닝

목 차

논문개요

Ⅰ. 서론 1

Ⅱ. 문제정의 4

Ⅲ. 관련연구 8
1. 빈발 항목집합 마이닝 8
2. Top-k 빈발 항목집합 마이닝 9
3. 높은 유틸리티 항목집합 마이닝 10
4. Top-k 높은 유틸리티 항목집합 마이닝 11

Ⅳ. 제안 방법론 12
1. 기반 자료구조 12
2. TKUL-Miner 알고리즘 15

Ⅴ. 실험 결과 27
1. 실행시간 비교 28
2. 메모리 사용량 비교 35

Ⅵ. 결론 및 향후 연구 39

참고문헌
ABSTRACT

표 목 차

[표 1] 항목 가격 ········· 4
[표 2] 트랜잭션 데이터베이스 ········· 4
[표 3] 트랜잭션 유틸리티(TU) ········· 7
[표 4] 1-항목집합 TWU ········· 7
[표 5] 실험 데이터집합의 특성 ········· 27

그 림 목 차

[그림 1] 항목집합 {d}와 {dc}의 노드와 유틸리티-리스트 ··············· 13

[그림 2] TKUL-Miner 알고리즘 ··············· 16

[그림 3] TKUL-FistLevelSearch 알고리즘 ··············· 18

[그림 4] Utility-List Join 함수 ··············· 21

[그림 5] TKUL-Search 알고리즘 ··············· 23

[그림 6] 전체 항목집합 트리 ··············· 24

[그림 7] Chain 데이터에 대한 실행시간 ··············· 30

[그림 8] Retail 데이터에 대한 실행시간 ··············· 30

[그림 9] Mushroom 데이터에 대한 실행시간 ··············· 32

[그림 10] Accidents 데이터에 대한 실행시간 ··············· 32

[그림 11] Chess 데이터에 대한 실행시간 ··············· 33

[그림 12] T10I4D100K 데이터에 대한 실행시간 ··············· 34

[그림 13] T40I10D100K 데이터에 대한 실행시간 ··············· 34

[그림 14] Chain 데이터에 대한 메모리 사용량 ··············· 36

[그림 15] Chess 데이터에 대한 메모리 사용량 ··············· 37

[그림 16] T10I4D100K 데이터에 대한 메모리 사용량 ··············· 37

[그림 17] T40I10D100K 데이터에 대한 메모리 사용량 ··············· 38

I. 서 론

항목집합 마이닝은 데이터마이닝 기법 중 연관규칙 탐사에서 항목들 간의 상관관계를 찾는 패턴 탐사 기술을 의미한다. 이 항목집합 마이닝의 초창기 방법론인 빈발 항목집합 마이닝(Frequent Itemset Mining, FIM)[1, 2]은 트랜잭션에서 항목의 존재 유무로 지지도를 계산하여 최소 지지도 이상의 빈도수를 나타내는 패턴을 찾았다. 이 최소 지지도는 사용자에 의해 주어지는데 이 값을 올바르게 설정하는 것은 상당히 어렵다. 그 이유는 같은 최소 지지도 값도 데이터에 따라 지나치게 크거나 작을 수 있기 때문이다. 만약 최소 지지도를 너무 크게 설정하면 결과 빈발 항목집합이 하나도 나오지 않게 되고, 반대로 너무 작으면 빈발 항목집합이 너무 많아 계산 시간과 메모리 사용 면에서 비효율적이다. 때문에 알맞은 최소 지지도를 설정하기 위해서는 그 값을 찾을 때까지 여러 번 다양한 값을 넣고 실행해 보아야 하는데 이는 많은 비용을 초래한다[3-5].

이 문제를 해결하기 위해 등장한 기법이 top-k 빈발 항목집합 마이닝(Top-k FIM)[3-5]이다. Top-k FIM은 최소 지지도 대신 사용자에게 보다 직관적이고 친숙한 값인 항목집합의 개수 k를 입력 받아 지지도 값을 기준으로 상위 k개의 빈발 항목집합을 찾는 것이다. 이 결과를 탐색하기 위해 top-k FIM은 0으로 초기화된 최소 지지도와 k개의 빈발 항목집합을 저장할 수 있는 고정 리스트를 이용한다. 그리고 먼저 찾은 항목집합들을 고정 리스트에 저장한 후 그 중 가장 작은 지지도 값으로 최소 지지도를 설정하고 이 최소 지지도 이상의 값을 갖는 항목집합을 발

견할 때 마다 리스트에 저장하며 최소 지지도 갱신을 반복하여 사용자가 원하는 패턴의 개수만큼의 결과를 출력한다.

빈발 항목집합 마이닝에 대한 연구는 항목의 중요도를 고려하는 가중치 빈발 항목집합 마이닝(Weighted FIM)[6, 7] 단계로 발전하여 높은 유틸리티 항목집합 마이닝(High Utility Itemset Mining, HUIM) [8-13]으로 진행되었다. 높은 유틸리티 패턴 마이닝은 이전의 연구와 비교해 실제 환경을 더욱 반영할 수 있도록 항목의 수량과 가중치를 동시에 고려하는 유틸리티의 개념을 도입한 마이닝 기법이다. 이와 유사하게 top-k 빈발 항목집합 마이닝에 대한 연구 또한 top-k 높은 유틸리티 항목집합 마이닝(Top-k HUIM)으로 진전되었다. Top-k 항목집합 유틸리티 마이닝은 데이터 스트림 top-k 높은 유틸리티 패턴 마이닝 [14, 15], 순차 top-k 높은 유틸리티 항목집합 마이닝[16], 트랜잭션 데이터베이스 top-k 높은 유틸리티 항목집합 마이닝[17, 18] 등 다양한 데이터에 대한 유틸리티 패턴 마이닝에 적용되고 있다.

트랜잭션 데이터베이스에서의 top-k 높은 유틸리티 패턴 마이닝은 의사 결정자가 원하는 높은 유틸리티 항목집합의 수 k로 상위 k개의 높은 유틸리티 항목집합을 찾는다. Top-k 높은 유틸리티 패턴 마이닝은 결과 항목집합을 마이닝하는 과정에서 0으로 초기화 되어있던 최소 유틸리티를 점차적으로 증가시킨다. 한 항목의 유틸리티는 항목의 수량과 가격의 곱으로 계산되기 때문에 단조감소 성질이 없고 이로 인해 일반 높은 유틸리티 항목집합 마이닝도 가지치기가 상당히 어려운데, top-k HUIM은 최소 유틸리티를 0으로 초기화하고 마이닝을 시작하기 때문에 탐색 공간의 가지치기는 더욱 제한적이다. 이와 마찬가지로 이 성질은 top-k HUIM의 유망하지 않은 항목집합을 가지치기 하는 것에서도 상당히 제

한적인 상황이 된다. 최소 유틸리티가 처음에 지정되지 않은 top-k 높은 유틸리티 항목집합 마이닝에서는 더 많은 후보를 생성할 것이고 이는 메모리 사용면에서 더욱 비용을 요구한다[17].

본 논문은 top-k HUI 마이닝의 비용을 줄이고 보다 효율적으로 높은 유틸리티 항목집합을 탐색하기 위해 새로운 알고리즘, TKUL-Miner (Top-k Utility-List based HUI-Miner)를 제안한다. 본 논문이 기여하는 것은 다음과 같다:

- 새로운 유틸리티-리스트 구조 기반의 top-k HUI 마이닝 프레임워크를 제안한다.
- 새로운 유틸리티-리스트 구조를 제안하고 최소 유틸리티를 빠르게 증가시키기 위한 전략들을 제안한다.
- 실험을 통해 본 알고리즘이 최근 top-k 알고리즘들[17, 18]과 비교하여 실행시간과 사용 메모리 면에서 성능의 개선을 보였고, 특히 조밀하고 길이가 긴 트랜잭션이 많은 데이터의 경우 더욱 우수함을 보였다.

논문의 구성은 다음과 같다. 2장에서 문제와 기존의 개념들에 대해 정의하고 관련 연구들에 대해 검토한다. 3장에서는 본 연구에서 새롭게 제안하는 자료 구조와 제안 알고리즘의 방법론을 설명한다. 제안하는 알고리즘의 실험적 성능은 유사 알고리즘들의 성능과 비교하여 4장에서 보여준다. 마지막으로 5장에서는 본 논문의 요약과 함께 향후 연구로 결론을 맺는다.

Ⅱ. 문제 정의

이 장에서는 top-k 높은 유틸리티 항목집합 마이닝을 위한 기본 개념과 관련된 정의를 설명한다[19, 25]. 항목들의 집합 $I = \{i_1, i_2, \cdots, i_m\}$은 항목들의 집합이고 $D = \{T_1, T_2, \cdots, T_n\}$는 트랜잭션 데이터베이스이다. 여기에서, 각 트랜잭션 T_d는 고유한 번호 d로 구분되며 이를 Tid라고 지칭한다. 한 항목 $i_p (1 \leq p \leq m)$의 수량과 단위 가격은 각각 $q(i_p)$와 $pr(i_p)$로 표기한다. 이는 [표 1]과 [표 2]에 나타나있다. 한 항목집합 X는 I의 부분집합인 고유한 항목들의 집합이다.

[표 1] 항목 가격

항목	a	b	c	d	e
가격	2	3	1	4	5

[표 2] 트랜잭션 데이터베이스

Tid	트랜잭션
T_1	(a, 1) (b, 3) (d, 1) (e, 1)
T_2	(a, 2) (c, 4) (d, 2)
T_3	(b, 2) (c, 1) (d, 1) (e, 2)
T_4	(b, 1) (c, 3) (d, 2) (e, 1)
T_5	(a, 2) (c, 5) (e, 1)
T_6	(a, 1) (b, 1) (c, 3) (d, 1) (e, 1)
T_7	(a, 3) (c, 2) (e, 2)

정의 1. 한 트랜잭션에서 항목의 유틸리티 u(i_p, T_d)는 아래와 같이 정의한다.

$$u(i_p, T_d) = q(i_p, T_d) \times pr(i_p) \tag{1}$$

정의 2. 한 트랜잭션에서 항목집합의 유틸리티 u(X, T_d)는 아래와 같이 정의한다.

$$u(X, T_d) = \sum_{i_p \in X \wedge X \subseteq T_d} u(i_p, T_d) \tag{2}$$

예를 들어 Table 1과 2로 u({bd}, T_1)을 구하면, u({bd}, T_1) = u({b}, T_1) + u({d}, T_1) = (3×3 + 1×4) = 13.

정의 3. 항목집합의 유틸리티 u(X)는 아래와 같이 정의한다.

$$u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d) \tag{3}$$

예를 들어, Table 1과 Table 2로 u({bd})를 구하면, u({bd}) = u({bd}, T_1) + u({bd}, T_3) + u({bd}, T_4) + u({bd}, T_6) = 13 + 10 + 11 + 7 = 41. 이와 마찬가지로 u({b}), u({bc}), u({bcd})을 계산하면 각 항목집합의 유틸리티가 각각 21, 19, 35 임을 계산할 수 있다.

이 결과와 같이 유틸리티 항목집합 마이닝에서는 항목집합의 길이가 늘어날 때 유틸리티 값이 증가하기도 하고 감소한다. 따라서 downward closure 성질[1]을 적용한 가지치기 전략을 사용할 수 없다[5].

정의 4. Top-k 높은 유틸리티 항목집합: 항목집합 X는 트랜잭션 데이터베이스 D에 대해 X의 유틸리티 값보다 큰 유틸리티 값을 갖는 항목집합의 수가 k개 보다 작으면 top-k 높은 유틸리티 항목집합이라고 지칭한다. 따라서 top-k HUI 들을 집합 H라고 할 때 H의 항목집합 수는 k보다 작거나 같다.

정의 5. 한 트랜잭션의 트랜잭션 유틸리티 TU(T_d)는 아래와 같이 정의한다.

$$TU(T_d) = \sum_{i_p \in T_d} u(i_p, T_d) \tag{4}$$

예를 들어, TU(T_1) = u({a}, T_1) + u({b}, T_1) + u({d}, T_1) + u({e}, T_1) = 20. 다른 트랜잭션들의 TU는 [표 3]에 계산되어있다.

정의 6. 한 항목집합의 트랜잭션 가중치 유틸리티(Transaction Weighted Utility, TWU) TWU(X)는 아래와 같이 정의한다.

$$TWU(X) = \sum_{X \subseteq T_d \wedge T_d \in D} TU(T_d) \tag{5}$$

따라서 TWU(X)는 항목집합 X가 포함된 모든 트랜잭션의 유틸리티로 해당 항목이 가질 수 있는 유틸리티의 최댓값이다. 그러므로 X의 실제 유틸리티는 항상 TWU(X)와 같거나 작다[6]. 예시 데이터에 대한 다른 항목들의 TWU 값은 [표 4]에 계산되어있다.

성질 1. TWDC(Transaction-Weighted Downward Closure)[13] 성질은 TWU 값에 항목집합이 하나 늘어날 때 마다 해당 항목집합이 포함된 트랜잭션 수가 항상 항목을 추가하기 전보다 작거나 같다는 단조 감소 성질이 있음을 의미한다.

예를 들면, TWU({b}) = TU(T_1) + TU(T_3) + TU(T_4) + TU(T_6) = 20 + 21 + 19 + 17 = 77 이고 {b}의 superset들은, TWU({ab}) = 37, TWU({bc}) = 57, 그리고 TWU({bcd}) = 57의 TWU 값을 갖는다. 이는 항목집합 X의 어떤 superset도 TWU 값으로 X의 TWU 값 보다 큰 값을 가질 수 없음을 보인다.

[표 3] 트랜잭션 유틸리티 (TU)

트랜잭션	T_1	T_2	T_3	T_4	T_5	T_6	T_7
TU	20	16	21	19	14	17	18

[표 4] 1-항목집합의 TWU

항목	a	b	c	d	e
TWU	85	77	105	93	109

Ⅲ. 관련 연구

1. 빈발 항목집합 마이닝(FIM)

연관규칙 탐사 알고리즘 Apriori[1]는 generate-and-test 방법을 이용하여 유용한 규칙을 마이닝 한다. 즉, 이 알고리즘은 데이터베이스를 여러 번 읽어 후보 패턴들을 나열해 빈발한 항목집합들을 찾는다. 첫 번째 스캔에서는 각 항목의 지지도를 계산하여 최소 지지도 이상의 지지도를 갖는 빈발 항목들을 찾는다. 그리고 그 항목들의 길이를 'level-wise' 방식으로 하나씩 늘려가며 다음 빈발 항목집합을 찾고 모든 빈발 항목집합들을 찾을 때 까지 이 작업을 반복한다. 다른 잘 알려진 빈발 항목집합 마이닝 알고리즘인 FP-Growth[2] 알고리즘은 전역 FP-트리(global FP-tree)를 구축한 후 재귀적으로 조건 트리를 생성하여 후보 항목집합의 생성 없이 패턴을 마이닝 한다. FP-트리는 먼저 데이터를 읽어 각 항목의 지지도를 계산하고 다시 데이터를 스캔할 때는 각 트랜잭션의 항목들을 지지도 내림차순으로 정렬하여 트리에 삽입해 항목의 공유를 최대화한다. 이 두 알고리즘은 지지도의 단조 감소 성질을 이용하여 패턴의 길이를 늘려가며 항목집합을 탐색한다. 그리고 이 성질을 이용해 지지도 값이 최소 지지도 보다 작은 항목집합은 더 이상 확장시키거나 성장시키지 않도록 하여 탐색 공간을 가지치기 한다. 하지만 이들은 알맞은 최소 지지도 값을 결정하는 것이 어렵다는 문제가 있다.

2. Top-k 빈발 항목집합 마이닝(Top-k FIM)

Itemset-Loop과 Itemset-iLoop[3]은 빈발 항목집합 마이닝의 최소 지지도 설정 문제를 해결하고자 제안된 알고리즘이다. 이 알고리즘들은 사용자가 원하는 빈발 항목집합의 개수 N과 결과 항목집합의 최대 길이인 Max_length를 입력 값으로 받는다. Itemset-Loop과 Itemset-iLoop 알고리즘은 이 값들을 사용하여 Apriori 기법에 backtracking 방법론을 적용하여 Max_length 이하의 길이를 갖는 항목집합들을 지지도 내림차순으로 N개 탐색한다. 알고리즘 TFP[4]도 같은 방법론을 따르지만 이는 Itemset-Loop 알고리즘과 달리 결과 항목집합의 최소 길이인 min_l을 입력받아 min_l 길이 이상인 동시에 지지도 기준으로 상위 k개인 항목집합들을 출력한다. 또한 TFP 알고리즘은 결과 항목집합 탐색을 위해 FP-Tree를 구축하여 빠르게 최소 지지도를 증가시키는 방법을 사용한다. TFP 알고리즘의 발전된 버전인 TF2P- Growth[5] 알고리즘은 사용자로부터 어떤 임계값도 입력받지 않고 빈발 항목집합들 마이닝해 사용자의 요청에 따라 즉각적으로 결과를 출력한다. MTK[3] 알고리즘은 메모리 제약을 두고 top-k 빈발 항목집합 마이닝을 ‘level-wise’ [1]로 마이닝하는 방식을 따른다. 따라서 이 알고리즘은 길이 1의 후보를 생성하고 패턴의 길이를 늘려가는 생성과 테스트(generate-and-test) 방법으로 마이닝을 수행한다. 이 알고리즘은 위 기법에 최소 지지도를 빠르게 증가시키며 탐색 공간을 가지치기하는 전략들을 추가하여 효율성과 메모리 제약으로 실전 응용에 장점을 갖도록 하였다.

3. 높은 유틸리티 항목집합 마이닝(HUIM)

높은 유틸리티 항목집합 마이닝은 내부 유틸리티와 외부 유틸리티 요소를 모두 고려해 수량과 가중치가 존재하는 실제 상황을 반영하기 위해 제안되었다. Two-Phase[8] 알고리즘은 'level-wise' 방법을 이용해 두 단계로 높은 유틸리티 패턴을 탐색한다. 첫 번째 단계에서 이 알고리즘은 TWU 값이 최소 유틸리티보다 작지 않은 항목집합들을 마이닝하고 두 번째 단계에서 데이터베이스를 다시 읽어 후보 항목집합들의 실제 유틸리티를 계산하여 결과를 찾는다. 알고리즘 IHUP[9]는 이 두 단계 기법을 FP-Growth와 같이 트리의 재귀 구축으로 패턴을 성장시키는 방법론에 적용한 알고리즘이다. 이 알고리즘은 TWU 내림차순으로 항목들을 삽입하여 효율적으로 트리를 구축하고 Two-phase 알고리즘과 같이 마지막에 후보들의 실제 유틸리티를 계산하여 결과 항목집합을 찾는다. UP-Growth와 UP-Growth+[10] 알고리즘은 IHUP 알고리즘을 개선하기 위해 추가적인 가지치기 전략들로 생성하는 후보 항목집합의 수를 줄였다. 특히 UP-Growth+ 알고리즘은 트리의 노드에 최소 노드 유틸리티를 저장시켜 보다 작은 과대평가 유틸리티 값을 계산해 후보의 수를 크게 줄이고 이로 인해 실행시간과 사용 메모리를 크게 줄였다.

이 후 연구는 실제 유틸리티를 유지하는 유틸리티-리스트 자료구조를 사용하여 마지막에 데이터베이스를 다시 읽는 과정을 생략시킨 HUI-Miner[11] 알고리즘이 제안되었다. 패턴 성장 기법의 높은 유틸리티 패턴 마이닝은 두 번째 단계에서 많은 실행시간을 소요하기 때문에 이 방법론은 기존의 방법론에서 속도를 크게 줄였다. 알고리즘 FHM[12]은 유틸리티-리스트 구조를 사용하면서 길이가 2인 항목집합의 TWU를 저

장한 EUCS 구조를 추가하여 마이닝의 탐색 공간을 가지치기해 HUI-Miner 알고리즘과 비교해 희소한(Sparse) 데이터에 대한 실행시간을 감소시켰다. 최근에 소개된 유틸리티-리스트 기반 HUIM 알고리즘인 TUL-Miner[13] 알고리즘은 항목집합들의 트랜잭션 유틸리티를 이용한 효과적인 탐색 공간 가지치기 전략을 제안하였고 공통 유틸리티를 활용하여 조인 연산의 계산 속도를 감소시켰다. 하지만 높은 유틸리티 항목집합 마이닝 알고리즘들은 근본적으로 적절한 최소 유틸리티를 결정하는 것이 어렵다는 한계가 있다.

4. Top-k 높은 유틸리티 항목집합 마이닝(Top-k HUIM)

TKU[17] 알고리즘은 UP-Growth 알고리즘을 기반으로 높은 유틸리티 항목집합에서 top-k 패턴을 찾기 위해 제안되었다. Top-k 높은 유틸리티 항목집합 마이닝 문제에서는 최소 유틸리티가 미리 지정되지 않기 때문에 생성되는 많은 수의 후보를 줄이기 위해 최소 유틸리티를 빠르게 증가시키는 다섯 가지 전략들을 적용하여 그 결과, 최적의 최소 유틸리티로 UP-Growth를 실행한 최적(optimal) 성능과 유사한 성능을 보였다. REPT[18] 알고리즘은 TKU 알고리즘에 최소 유틸리티를 증가시키는 전략들을 추가 및 변형시키고 사용 메모리의 효율성을 위해 보다 효율적인 자료구조를 제안하여 속도와 메모리 사용의 성능을 개선시켰다. 그러나 유틸리티는 단조 감소 성질이 없고 최소 유틸리티가 0에서부터 시작하기 때문에 top-k 높은 유틸리티 항목집합 마이닝을 실행하기 위한 시간과 메모리는 일반 높은 유틸리티 항목집합 마이닝보다 더 큰 비용을 요구한다.

Ⅳ. 제안 방법론

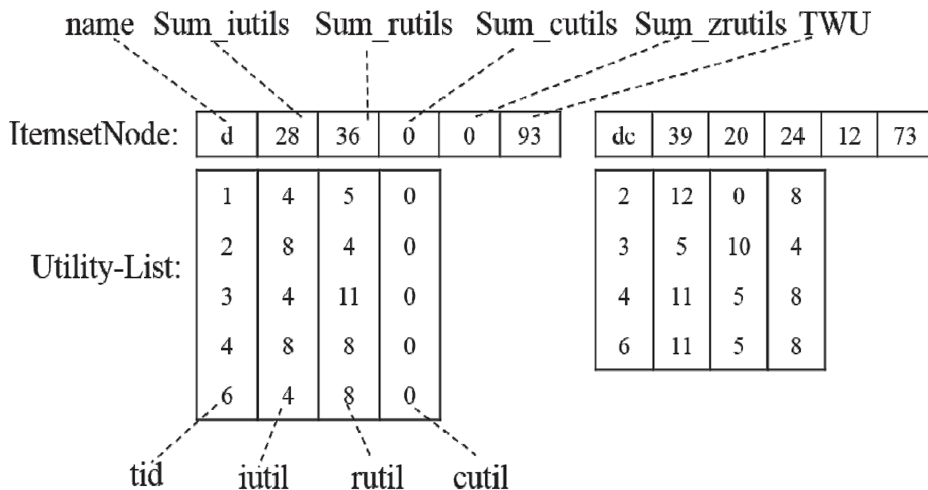
1. 기반 자료구조

1.1 항목집합 노드 구조

TKUL-Miner 알고리즘은 탐색 트리의 항목집합 노드를 새롭게 설계하여 다른 유틸리티-리스트 기반의 알고리즘들[11, 13]의 항목집합 노드보다 더 많은 정보를 저장한다. 따라서 이 항목집합 노드는 참고문헌 [11]에서 제안한 속성인 name, Sum_iutils, Sum_rutils와 참고문헌 [13]에서 제안한 TWU 속성에 Sum_cutils 속성과 Sum_zrutils 속성을 추가하였다. 항목집합 노드의 모습은 [그림 1]의 예시와 같다. 속성 name은 노드의 이름을 의미하고 Sum_iutils는 해당 항목집합의 실제 유틸리티, Sum_rutils는 항목집합의 각 트랜잭션에서 해당 항목집합 뒤의 나머지 항목들의 유틸리티인 rutil 값들의 합을 의미하고 TWU 속성은 해당 항목집합의 TWU 값을 의미한다. 그리고 추가 속성인 Sum_cutils는 새로운 노드 생성 시 공통 유틸리티인 cutil 값[13]들의 합이고 Sum_zrutil은 항목집합을 포함하는 트랜잭션의 rutil이 0인 iutil 값들의 합이다.

1. 2 유틸리티-리스트 구조

모든 항목집합이 갖는 유틸리티 리스트 구조는 [그림 1]과 같이 기존의 유틸리티-리스트의 속성인 tid, iutil과 rutil 값을 갖고 UTU-List [13]의 cutil (common utility) 속성을 갖는다. Tid는 항목집합 X를 포함하는 트랜잭션의 아이디, iutil은 한 트랜잭션에서의 항목집합의 유틸리티인 u(X, T_d), 그리고 rutil은 한 트랜잭션에서 해당 항목집합 이후의 나머지 항목들 T/X의 유틸리티 값으로 $\sum_{i \in (T_d/X)} u(i, T_d)$를 계산하여 각각의 속성에 저장한다. 속성 cutil은 부모 노드의 iutil 값으로 유틸리티-리스트의 조인 연산에서 항목집합의 유틸리티를 계산할 때 사용된다. 모든 1-항목집합의 cutil은 0으로 초기화 된다.



[그림 1] 항목집합 {d}와 {dc}의 노드와 유틸리티-리스트

[그림 1]을 보면 항목집합 {d}를 포함하는 트랜잭션의 번호가 tid 속성에 순서대로 저장되고 각 트랜잭션에서 항목집합 {d}의 유틸리티가 그 다음 칼럼인 iutil 속성으로 저장되어있다. 그리고 그 트랜잭션을 TWU 오름차순으로 정렬했을 때 {d} 다음에 오는 항목들의 유틸리티 합을 rutil 칼럼에 저장하고 {d}는 1-항목집합이므로 cutil 칼럼은 모두 0으로 초기화 하였다. 그리고 각 iutil의 합을 항목집합 노드의 Sum_iutils 값으로, rutil의 합을 Sum_rutils 값으로, cutil의 합을 Sum_cutils로 저장한다. 그리고 rutil이 0인 트랜잭션의 iutil 합을 Sum_zrutil에 저장해야 하지만, 항목집합 {d}는 rutil이 0인 트랜잭션이 없으므로 0을 저장한다. 그리고 각 트랜잭션의 TU의 합을 TWU로 저장한다. 항목집합 {dc}도 같은 방법으로 저장하는데 이 때 rutil은 {d}보다 뒤에 나타나는 항목집합 {c}의 rutil을 해당 트랜잭션의 rutil 칼럼에 그대로 저장한다. 그리고 {dc}의 경우 Tid가 2인 트랜잭션의 rutil이 0이므로 해당 트랜잭션의 iutil인 12를 Sum_zrutil로 저장한다. 또한 항목집합 {dc}는 {d} 항목집합의 다른 자식 노드들과 {d}라는 공통 항목을 가지므로 {dc}에 cutil 값으로 항목집합 {d}의 유틸리티-리스트에서 해당 트랜잭션의 iutil 값을 저장한다.

2. TKUL-Miner 알고리즘

2.1 제안 알고리즘의 마이닝 방법

본 장에서는 TKUL-Miner 알고리즘의 효율적인 top-k HUI 탐사 방법에 대해 설명한다.

마이닝하기에 앞서 트랜잭션 데이터베이스와 각 항목의 가격, 그리고 사용자의 지정 값인 k가 주어진다. 그리고 0으로 초기화한 최소 유틸리티인 minutil은 top-k 높은 유틸리티 항목집합을 마이닝하는 동안 점차 증가하게 된다. 제안 알고리즘은 [그림 2]에 나타나있는 사전 작업으로 시작된다.

TKUL-Miner 알고리즘은 마이닝 과정 동안 사용할 초기 유틸리티-리스트와 EUCS[12] 구조를 구축하기 위해 데이터베이스를 두 번 스캔한다. 기본 구조들을 구축하는 사전 작업을 마친 후 TKUL-Miner 알고리즘은 본격적인 top-k 높은 유틸리티 항목집합 마이닝을 시작한다. 알고리즘은 항목의 유틸리티-리스트인 초기 유틸리티-리스트의 조인연산으로 각 항목집합의 항목을 확장시키며 top-k 높은 유틸리티 항목집합을 탐색한다. 그리고 한 항목집합의 유틸리티가 minutil 보다 작지 않을 때 마다, 그 항목집합을 최소 힙에 삽입한다. 그리고 최소 힙이 k개의 항목집합으로 꽉 차있으면서 힙 안의 항목집합 중 유틸리티가 가장 작은 것의 값이 minutil 보다 크면, minutil을 그 최소 유틸리티 값으로 갱신한다. 이 과정은 더 이상 생성할 항목집합이 없을 때까지 진행된다. 본 논문은 최소 유틸리티를 빠르게 증가시켜 효과적으로 탐색 공간을 가지치기하고 실행시간을 줄이기 위해 FSD, FUZ, FCU 전략을 적용한다.

Algorithm: TKUL-Miner Algorithm

input : D, a transaction database; minutil; k.

output: the top-k high-utility itemsets.

1 Scan D to calculate the TWU of 1-itemsets

2 I* ← each item i such that TWU(i) ≥ minutil

3 Sort items in TWU ascending values on I*

4 Scan D to build the initial utility-list of each item i ∈ I* and build the EUCS structure

5 TUL-FirstLevelSearch(∅, I*, minutil, EUCS, k)

[그림 2] TKUL-Miner 알고리즘

세부적으로는 사전작업에서 효과적으로 자료구조를 생성하기 위해 1-항목집합의 재배치를 수행한다. 먼저 알고리즘은 [그림 2]의 첫 번째 줄과 같이 데이터베이스를 읽어 각 1-항목집합의 TWU를 계산한다. 그 다음에 TWU 값이 minutil 보다 크거나 같은 항목들에 대해서 TWU 오름차순으로 정렬한다. 앞서 말한 예시 데이터에서 1-항목집합은 b < a < d < c < e 순으로 정렬된다. 이 재배치 과정을 마치면, 데이터베이스를 다시 읽으며 초기 유틸리티-리스트와 EUCS 구조를 구축한다. 이 때 읽는 데이터베이스의 트랜잭션은 TWU 오름차순으로 정렬한 1-항목집합과 동일하게 재배치 한 후 유틸리티-리스트에 각 정보를 저장한다.

TKUL-Miner의 두 개의 주요 알고리즘은 TKUL-FirstLevelSearch와 TKUL-Search 이다. 위 사전 작업을 마치고, TKUL-Miner는 FSD와 RUZ 전략이 있는 TKUL-FirstLevelSearch를 호출하며 마이닝 작업을 시작한다.

2.2 FSD (First-level Search in TWU Decreasing-order) 전략

TKUL-FirstLevelSearch 알고리즘은 [그림 3]의 첫 번째 줄에 탐색 트리의 첫 번째 레벨인 1-항목집합을 TWU 내림차순으로 탐색하는 FSD 전략을 사용한다. 이는 HUI-Miner[11]와 FHM[12] 알고리즘이 TWU 오름차순으로 탐색하는 것과는 반대의 탐색 방법이다. TKUL-FirstLevelSearch 알고리즘은 특별히 1-항목집합에 대해서만 탐색하고 2-항목집합들만 생성한다. 그리고 나머지 항목집합들은 TKUL-Search 알고리즘이 탐색하고 생성한다. TWU 오름차순으로 나열한 초기 유틸리티-리스트는 오른쪽에 위치한 항목일수록 TWU 값이 크다. 그리고 TWU 값이 큰 항목집합의 유틸리티가 실제로 더 높은 값의 유틸리티를 가지는 자식을 생성할 가능성이 높다. 따라서 오른쪽 1-항목집합에 대한 패턴 확장을 우선적으로 실행하면 왼쪽에서부터 탐사할 때 보다 빠르게 최소 유틸리티인 minutil을 증가시킬 수 있고 이는 가지치기 효율을 더욱 개선한다.

```
Algorithm: TKUL-FirstLevelSearch Algorithm
input : P, an itemset; ExtensionsOfP, a set of extensions of P;
        minutil; EUCS; k.
output: minHeap, the top-k high-utility itemsets.
1   for i ← length(ExtensionsOfP)-1 to 0 do
2       Px ← ExtensionsOfP[i]
3       if Px.Sum_iutils ≥ minutil then
4           add Px to minHeap(k)
5       if Px.Sum_iutils + Px.Sum_rutils − Px.Sum_zrutils ≥ minutil then
6           ExtensionsOfPx ← ∅
7           for j ← i+1 to length(ExtensionsOfP)-1 do
8               Py ← ExtensionsOfP[j]  ▷ y after x
9               if c ≥ minutil such that (x, y, c) ∈ EUCS then
10                  Pxy ← Utility-List_Join (Px, Py, minutil)
11                  if Pxy.Sum_itus ≥ minutil then
12                      ExtensionsOfPx ← ExtensionsOfPx ∪ Pxy
13          TKUL-Search(Px, ExtensionsOfPx, minutil, k)
```

[그림 3] TKUL-FistLevelSearch 알고리즘

TKUL-FirstLevelSearch 알고리즘에서 가장 먼저 선택되는 항목집합은 P항목에 x를 확장시킨 항목 Px이며 가장 오른쪽에 위치한 항목집합이다. 그리고 이 알고리즘에서 Px는 가장 오른쪽에서부터 왼쪽으로 하나씩 탐사한다. 알고리즘은 선택된 Px에 대해 Px의 유틸리티인 Sum_iutils가 최소 유틸리티 minutil 보다 크면 Px는 top-k 높은 유틸리티 항목집합의 후보가 된다. [그림 3]의 네 번째 줄은 이 후보를 최소

힙에 저장하고, 힙이 꽉 찼을 때는 내부적으로 힙 안 항목집합의 최소 유틸리티 값으로 minutil을 갱신한다.

2.3 RUZ(Reducing the overestimated Utility by Sum_zrutils)전략

기존의 알고리즘들 [11, 12]에서는 각 항목집합의 과대평가된(overestimated) 유틸리티로 항목집합이 유망한지를 검사했다. 만약 우리가 그 과대평가된 유틸리티를 top-k 높은 유틸리티 항목집합들을 하나도 잃지 않으며 안전하게 줄일 수 있다면, 마이닝 과정이 보다 효율적일 것이다. Top-k 높은 유틸리티 항목집합 마이닝의 탐색 공간 가지치기를 더 개선시키기 위해, 이 과대평가된 유틸리티를 최소화 하는 RUZ 전략을 사용한다. RUZ 전략은 [그림 3]의 다섯 번째 줄과 같이 계산된다.

$$Px.Sum_iutils + Px.Sum_rutils - Px.Sum_zrutils \quad (6)$$

식 (6)은 이전의 Sum_iutils와 Sum_rutils의 합으로 계산된 과대평가된 유틸리티에서 Sum_zrutils 값을 제외시키는 계산을 한다. 이 방법은 항목집합 Px의 트랜잭션들 중 rutil 값이 0인 트랜잭션들은 Px의 자식에는 포함되지 않을 것이라는 사실에 기반 한다. 이 말은 곧 과대평가된 유틸리티에서 rutil이 0인 트랜잭션들의 iutil값이 제외되더라도 Px의 자손들이 가질 수 있는 유틸리티의 상한 값으로 유효하다는 의미가 된다. 따라서 만약 RUZ 전략에 의해 최소화된 항목집합 Px의 과대평가 유틸리티 값이 minutil 보다 작으면 유망하지 않은 항목으로 파악되어 가지치기 된다.

항목집합 Px가 유망하다면 항목집합 x의 TWU 보다 TWU 값이 더 큰 y를 P에 확장시킨 항목집합 Py를 선택한다. 그리고 EUCS 구조[12]를 통해 (x, y)값이 최소 유틸리티보다 작지 않은지 검사한다. 항목 x와 y의 EUCS 값이 유망하다면, [그림 3]의 열 번째 줄에서 Utility-List_Join 함수를 호출한다. 이 함수는 항목집합 Px와 Py의 유틸리티-리스트를 조인해 항목집합 Pxy와 Pxy의 유틸리티-리스트를 생성한다. 그 과정에서 이 함수는 [그림 4]의 열세 번째에서 스무 번째 줄의 코드와 같이 참고문헌 [13]의 DTJ (Decreasing TWU while Joining) 전략을 이용해 조인 중에 항목집합 Pxy가 유효하지 않다고 판단되면 연산을 중단시킨다. 유틸리티-리스트에 있는 트랜잭션의 유틸리티인 TU 값을 사용하기 위해, TKUL-Miner 알고리즘은 [표 3]을 이용해 해당하는 Px와 Py의 TWU에서 TU 값을 제외시킨다.

```
Function: Utility-List Join Function
input: Px, the extension of P with an item x;
       Py, the extension of P with an item y; minutil.
output: Pxy.
1   Pxy ← Px ∪ Py
2   rx ← Px.Sum_itus          ▷ TWU(Px)
3   ry ← Py.Sum_itus          ▷ TWU(Py)
4   i ← j ← 0
5   while i < Px.num and j < Py.num do
6       ex ← Px.utulist[i]
7       ey ← Py.utulist[j]
8       if ex.tid == ey.tid then
9           exy ← (ex.tid, ex.itu, ex.iutil+ey.iutil−ex.cutil, ey.rutil, ex.cutil)
10          Pxy.UTUList ← Pxy.UTUList ∪ {exy}
11          i++
12          j++
13      else if ex.tid < ey.tid then
14          rx −= ex.itu ▷ minus TU of the transaction not belong to Pxy
15          if rx < minutil then break    ▷ Pxy is definitely not a TKHUI
16          i++
17      else
18          ry −= ey.itu
19          if ry < minutil then break
20          j++
21  return Pxy
```

[그림 4] Utility-List Join 함수

Utility-List_ Join 함수의 연산을 통해 항목집합 Pxy를 생성하면 TKUL-FirstLevelSearch 알고리즘에서 그 다음 항목집합 Py가 더 이상 없을 때 까지 항목집합 Px에 각 Py를 조인하여 자식 노드를 생성한다. 그리고 TKUL-FirstLevelSearch 알고리즘은 각 Px의 모든 자식 노드들을 생성할 때 마다 TKUL-Search 알고리즘을 호출한다.

TKUL-Search 알고리즘은 TKUL-FirstLevelSearch 알고리즘과 마찬가지로, RUZ와 EUCS 전략과 같은 Utility-List_ Join 함수를 사용한다. 그러나 TKUL-Search 알고리즘은 가지치기 효율을 높이기 위해 TKUL-FirstLevelSearch 알고리즘과 다른 두 가지 특징을 가진다. 하나는 기존의 다른 유틸리티-리스트 구조 기반 알고리즘들과 마찬가지로 항목집합을 TWU 오름차순으로 탐색하여 효과적인 가지치기를 수행한다는 것이고, 다른 하나는 TKUL-Search 알고리즘이 유망하지 않은 항목집합을 빠르게 가지치기하기 위해 추가적으로 Sum_cutils 값을 이용한 FCU 전략을 가진다는 것이다. TKUL-Search 알고리즘은 [그림 5]에 나타나있다.

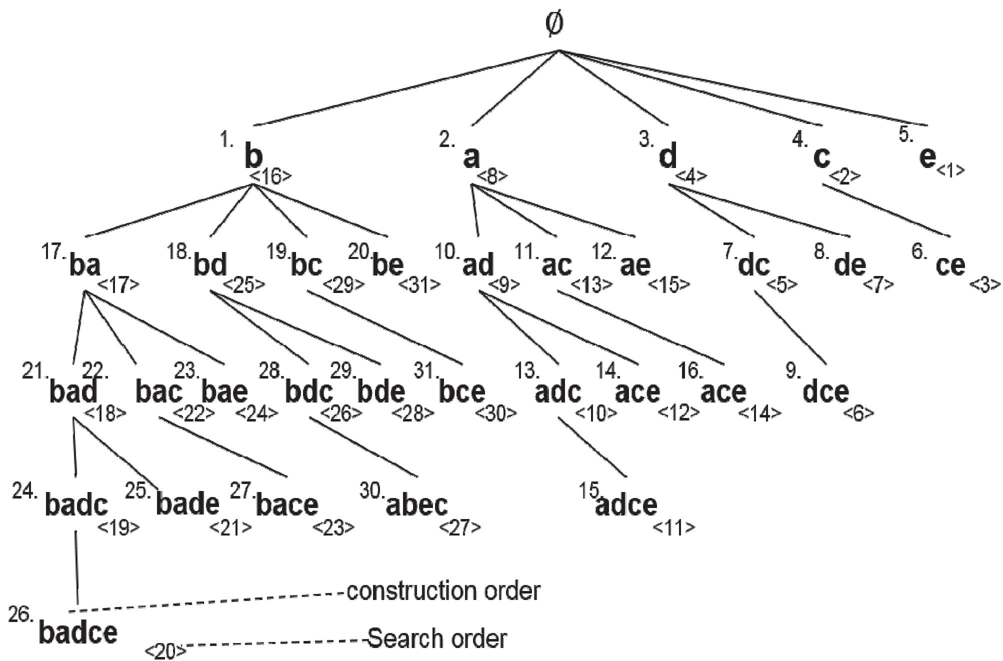
```
Algorithm: TKUL-Search Algorithm
input : P, an itemset; ExtensionsOfP, a set of extensions of P;
        minutil; EUCS; k.
output: minHeap, the top-k high-utility itemsets.
1    for i ← length(ExtensionsOfP)-1 to 0 do
2         Px ← ExtensionsOfP[i]
3         if Px_Sum_iutils ≥ minutil then
4              add Px to minHeap(k)
5         if (Px_Sum_iutils + Px.Sum_rutils - Px.zeroRutils) ≥ minutil then
6              ExtensionsOfPx ← ∅
7              for j ← i+1 to length(ExtensionsOfP)-1 do
8                   Py ← ExtensionsOfP[j]
9                   if (px.chNum > 0 and
                     c ≥ minutil such that (x, y, c) ∈ EUCS) or
                     (px.chNum == 0 and Px.Sum_iutils + Py.Sum_iutils +
                    Py.Sum_rutils - Px.Sum_cutils >= minutil) then
10                       Pxy ← Utility-List_Join (Px, Py, minutil)
11                       if Pxy.Sum_itus ≥ minutil then
12                            ExtensionsOfPx ← ExtensionsOfPx ∪ Pxy
13             TUL-Search(Px, ExtensionsOfPx, minutil, k)
```

[그림 5] TKUL-Search 알고리즘

[그림 5]의 첫 번째 줄에서 알 수 있듯이 TKUL-Search 알고리즘은 항목집합을 TWU 오름차순으로 Px를 선택하여 패턴을 탐색하고 자식을 생성한다. 그리고 재귀적으로 TKUL-Search 알고리즘을 호출하여 패턴

의 탐색하며 자식 노드를 생성한다.

예시 데이터에 대한 항목집합 노드의 전체 트리의 모습이 [그림 6]에 나타나있다. 각 노드의 왼쪽 상단에 위치한 숫자는 TKUL-Miner 알고리즘의 항목집합 생성 순서를 의미하고 노드의 오른쪽 하단에 위치한 각 괄호 안에 숫자는 항목집합의 탐색 순서를 의미한다. TKUL-FirstLevel Search 알고리즘이 항목집합 {ce}에서부터 {ba}에 이르는 모든 2-항목집합을 생성하고 그 아래 나머지 항목집합 {dce}부터 {badce}는 TKUL-Search 알고리즘에 의해 생성된다.



[그림 6] 전체 항목집합 트리

TKUL-Search 알고리즘에서는 항목집합 P의 가장 왼쪽의 자식이 먼저 항목집합 Px로 선택된다. 그리고 Px의 Sum_iutils가 top-k 높은 유틸리티 항목집합으로 유효한지 검사하고 RUZ 전략으로 유망한 항목집합인지를 결정한다. TKUL-Search 알고리즘의 FCU 전략은 Px가 자식이 없을 때 이용하고 그렇지 않으면 TKUL-FirstLevelSearch 알고리즘의 EUCS 전략을 사용해 항목집합 Py와의 조인 연산 수행을 결정한다.

2.4 FCU (First child pruning by using Sum_Cutil) 전략

FHM 알고리즘은 항목집합 Px와 Py를 조인하여 생성하는 자식 노드 Pxy를 생성하기 전에 Pxy의 어림값으로 항목 (x, y)의 EUCS 값이 minutil 보다 작지 않은지를 검사한다. 우리는 EUCS의 (x, y)로 계산되는 Pxy의 어림값을 줄이기 위해 유틸리티-리스트의 Sum_cutils라는 속성을 이용한다. FCU 전략은 Px의 첫 번째 자식에 대해서만, 아래 식을 이용해 계산의 결과 값이 minutil 보다 크거나 같을 때만 조인 연산을 수행하도록 하는 것이다.

$$Px.Sum_iutils + Py.Sum_iutils + Py.Sum_rutils - Px.Sum_cutils \quad (7)$$

이 식의 결과 값은 Pxy의 과대평가 값으로 항상 EUCS 보다 작거나 같은 값을 제공한다. 그러한 이유는 Px의 첫 번째 자식 항목집합 Pxy의 실제 유틸리티인 Sum_iutils는 본래 항목집합 Px와 Py의 공통 트랜잭션의 iutils의 합에 해당 cutils 값을 뺀 것이고, Pxy의 Sum_rutils의 최댓값은 Py의 Sum_rutils이기 때문이다. 따라서, Pxy의 새로운 과대평가

값이 minutil 보다 작으면 Px와 Py의 조인 연산이 무의미 할 것을 의미한다.

이 전략이 Px의 첫 번째 자식에 대해서만 생성 여부를 결정할 수 있는 이유는 이 전략의 계산 값이 해당 자식과 그 자식의 후손 노드들의 실제 유틸리티 보다 항상 크거나 같은 것은 참이지만 그 자식의 형제 노드들과 결합했을 때 생길 수 있는 다른 자식에 대해서는 더 작을 수도 있기 때문이다. 즉, 두 번째 자식 노드와 그 이후의 자식 노드들은 왼쪽 형제 노드들에게 영향을 주지만 첫 번째 자식은 이들과 달리 다른 형제 노드에게 영향을 주지 않기 때문에 FCU 계산 값이 Pxy의 상한 유틸리티 값으로 유효하다. 만약 FCU 전략을 두 번째와 그 이후의 자식들에게도 적용한다면 유망한 자식이 생성되지 않을 수 있다.

TKUL-Search 알고리즘은 위에 설명한 과정을 통해 유망하다고 판단되면 항목집합 Px와 Py가 확장된 항목집합 Pxy를 생성하도록 한다. 그리고 알고리즘은 Px에 모든 가능한 확장 항목 y에 대해 조인 연산을 수행한다. TKUL-Miner 알고리즘은 더 이상 유망한 항목집합이 없을 때까지 이 과정을 반복한다. 그리고 알고리즘이 종료되면 최소 유틸리티는 하나의 값으로 고정되고 사용자가 원하는 개수의 높은 유틸리티 항목집합이 최소 힙에 저장되어 있게 된다.

Ⅴ. 실험 결과

[표 5] 실험 데이터집합의 특성

데이터명	크기 (KB)	트랜잭션 수	항목 수	평균 길이	최대 길이	데이터 분포정도
Accidents	59663	340183	468	33.8	51	Dense
Chain	63573	1112949	46086	7.3	170	Sparse
Chess	591	3196	75	37	37	Dense
Mushroom	963	8124	119	23	23	Dense
Retail	6076	88162	16470	10.3	76	Mid-Sparse
T10I4D100K	6268	98424	1000	10.1	30	-
T40I10D100K	24905	100000	1000	39.6	78	-

본 논문에서 제안하는 알고리즘 TKUL-Miner가 높은 유틸리티 항목 집합들을 효율적으로 탐사하는 것을 보여주기 위하여 top-k 높은 유틸리티 패턴 마이닝 알고리즘인 TKU[17]와 REPT[18] 그리고 일반 높은 유틸리티 패턴 마이닝 알고리즘 UP-Growth^{+}[10]와 성능을 비교 평가하였다. REPT 알고리즘에 대해서는 RSD 구조를 구축하기 위해 필요한 값 N을 해당 논문 [18]을 따라 설정하였다. UP-Growth^{+}는 일반 높은 유틸리티 패턴 마이닝 알고리즘이므로 주어진 k 값으로 top-k 높은 유틸리티 패턴 마이닝 알고리즘을 실행했을 때 계산 된 최소 유틸리티 값인 최적(optimal) 최소 유틸리티 임계값으로 프로그램을 실행하였다.

성능 평가에 사용된 데이터는 NU-MineBench 2.0[19]에서 제공하는 실 환경 데이터 Chain과 FIM Repository[20]에서 제공하는 실 환경 데이터인 Accidents와 Chess, Mushroom, Retail을 사용하였다. 그리고 생성 데이터는 IBM Quest Data Generator[1]로 생성한 데이터인 T10I4D100K와 T40I10D100K을 사용하였다. 데이터 Chain은 항목의 수량과 가중치 값이 주어진 데이터이지만 나머지 실 환경 데이터는 수량과 가중치 값이 주어지지 않아 생성 데이터와 같이 수량은 1부터 10까지의 랜덤 수를 생성하였고, 항목가격은 1에서 1000까지의 단위 가격으로 로그 정규 분포를 따르도록 하였다.

실 환경 데이터는 트랜잭션의 길이가 서로 유사해 평균길이 부근에 조밀(dense)한 분포를 가지거나 혹은 트랜잭션의 길이가 서로 상당히 달라 상당히 퍼져있는 희소(sparse)한 분포를 가진다. 그러나 생성 데이터 T10I4D100K와 T40I10D100K은 랜덤으로 생성된 데이터이므로 데이터의 형태가 상대적으로 고른 분포를 가진다.

본 실험은 MS Windows 7 64bit 운영체제의 Intel i7-4770 CPU 3.40GHz와 32GB 메모리의 환경에서 실행하였고, 모든 프로그램은 Microsoft Visual Studio 2010 에서 C++로 구현하여 그 실험 결과를 측정하였다.

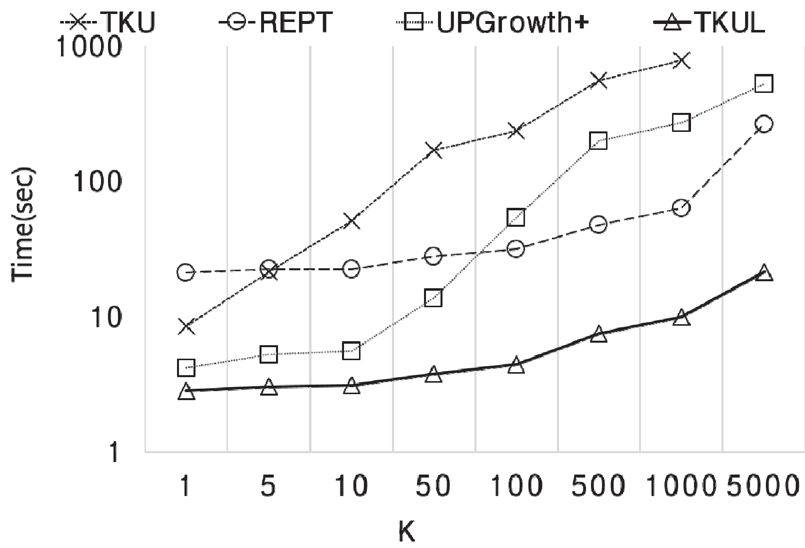
1. 실행시간 비교

이 절에서는 각 실 환경 데이터와 생성 데이터에 대해 기존 알고리즘들인 TKU, REPT, UP-Growth^{+}(optimal)와 제안 알고리즘 TKUL-Miner의 실행시간을 비교한 실험 결과를 보인다. 이 실험의 결과는 로그

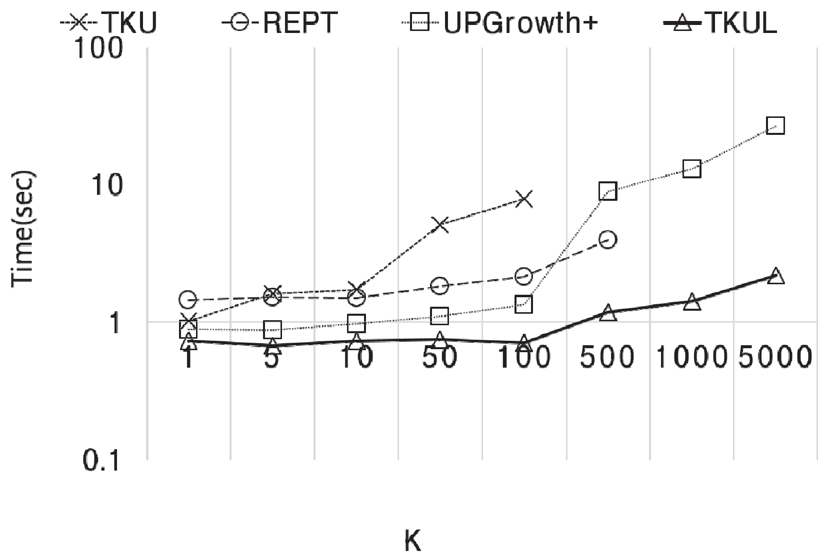
눈금 간격으로 초 단위를 표현하였고 알고리즘의 실행시간이 10,000 초를 넘기면 프로그램을 종료하였다.

[그림 7]은 희소한 분포를 가진 실 환경 데이터 Chain에 대한 알고리즘의 실행시간을 보여준다. 이를 따르면 제안 알고리즘인 TKUL-Miner 알고리즘은 늘어나는 k값에 대해 언제나 가장 빠르다. 그리고 최적의 최소 유틸리티 값으로 마이닝을 한 UP-Growth^{+} 알고리즘이 k 가 50 이하일 때까지 두 번째로 빠르고 k가 100 이상일 때부터 REPT (N = 1000)이 두 번째로 빠르다. 제안 알고리즘과 두 번째로 빠른 알고리즘의 차이를 보기 위해 k가 5000일 때는 비교하면, TKUL-Miner 알고리즘의 실행시간이 약 21초인데 반해 REPT 알고리즘의 실행시간은 약 266초를 기록한다. 이를 통해 제안 알고리즘이 기존 알고리즘을 약 10배 개선시켰음을 알 수 있다. 한편 초창기 top-k 높은 유틸리티 항목집합 마이닝 알고리즘인 TKU는 대부분의 경우 가장 느리고 k가 5000이 되었을 때는 그 실행시간이 10000초를 넘어 그래프 상에 점을 표시하지 않았다.

희소한 분포에 가까우면서 Chain 데이터 보다 평균 트랜잭션의 길이가 조금 더 긴 실 환경 데이터 Retail에 대한 알고리즘의 실행시간은 [그림 8]에 나타나있다. 이 데이터에 대한 실험에서도 제안 알고리즘은 TKUL-Miner의 실행 시간이 가장 적게 걸린다. 반면 재귀적으로 트리를 생성하면서 패턴을 성장시키는 알고리즘들의 성능이 k에 따라 실행시간이 기하급수적으로 늘어나 k가 커지면서 두 top-k 높은 유틸리티 항목집합 마이닝 알고리즘(REPT의 N = 1000)은 10,000 초 이상 소요되었다. 이로서 희소한 데이터에 대해 제안 알고리즘의 실행 속도가 기존의 알고리즘을 개선했음을 알 수 있다.



[그림 7] Chain 데이터에 대한 실행시간

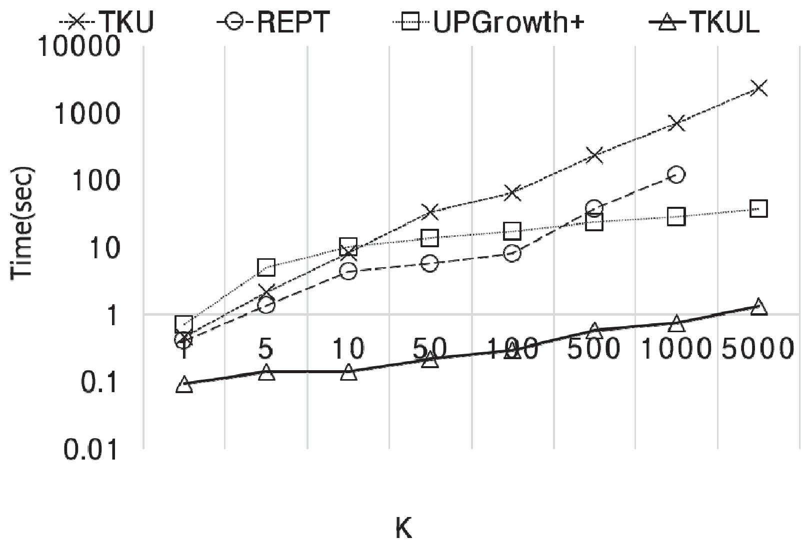


[그림 8] Retail 데이터에 대한 실행시간

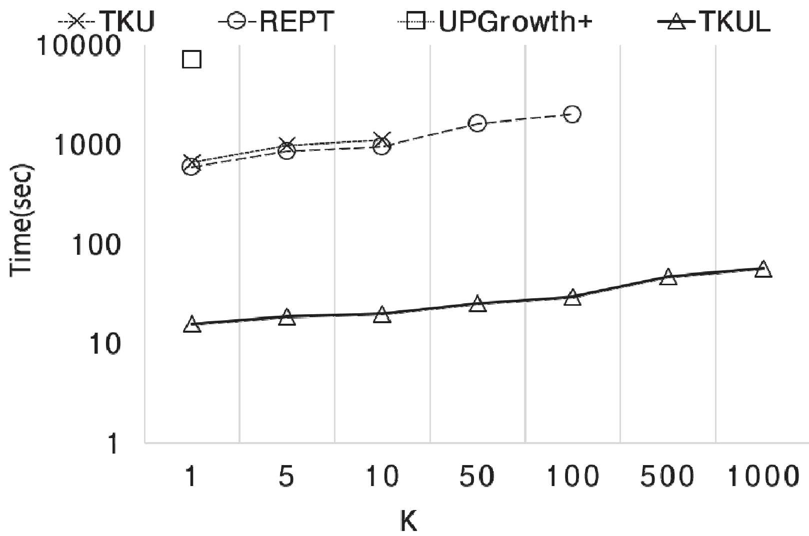
[그림 9]는 조밀한 분포를 가진 Mushroom 데이터에 대한 각 알고리즘의 실행 속도 실험 결과이다. 이를 보면 제안 알고리즘과 기존 알고리즘들의 속도 차이가 희소한 분포를 가진 데이터에 대한 실험 결과 보다 크다는 것을 알 수 있다. TKUL-Miner 알고리즘은 실험에서 어떤 k 값에서도 2초를 넘지 않는다. REPT 알고리즘(N = 100)의 성능과 비교하면 제안 알고리즘은 k가 50일 때 약 20배 더 빠르고 k가 1000일 때 약 160배 더 빠르다.

마찬가지로 조밀한 분포를 가지고 Mushroom 데이터와 비교해 데이터의 크기가 상당히 큰 데이터 Accident에 대한 실험 결과인 [그림 10]를 보면 제안 알고리즘의 성능이 다른 알고리즘들과 비교해 상당히 크며 k가 100일 때 제안 알고리즘의 속도는 REPT(N = 200) 알고리즘의 속도 보다 약 30배 빠르다. 그리고 이 데이터에 대한 실험에서 최적의 유틸리티로 실험한 UP-Growth^{+}알고리즘의 성능이 가장 좋지 않은데 그것은 이 알고리즘이 일반 높은 유틸리티 항목집합 마이닝에 최적화 되어있는 알고리즘이며 데이터의 크기가 크고 조밀한 데이터에 대해 해당 알고리즘의 메모리가 너무 많이 소요되어 그로 인한 시간이 더욱 길어졌기 때문이다.

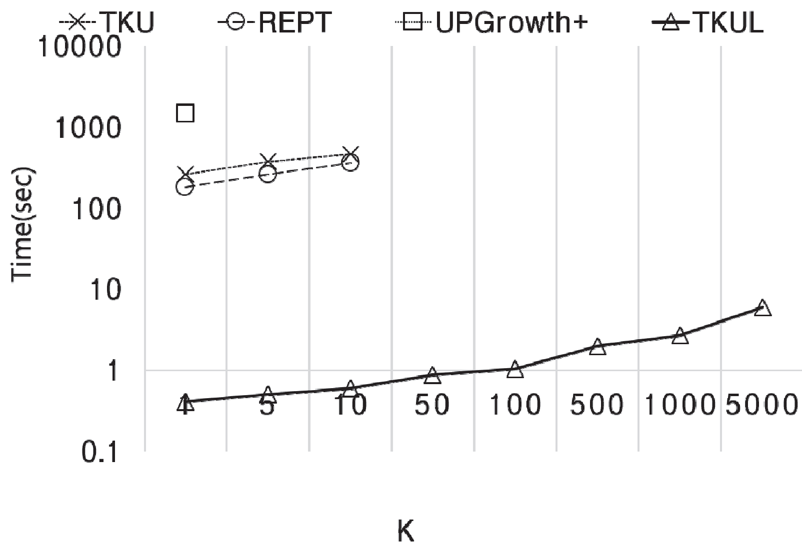
조밀한 분포의 실 환경 데이터 Chess에 대한 실험 결과인 [그림 11] 또한 [그림 10]과 비슷한 결과를 보인다. Chess 데이터는 트랜잭션의 평균길이가 가장 길어 REPT(N = 100)알고리즘을 포함한 패턴 성장 기법을 기반으로 하는 알고리즘들의 속도가 상당히 나쁘게 나타나며 평균적으로 제안 알고리즘이 약 1000배 빠르다.



[그림 9] Mushroom 데이터에 대한 실행시간

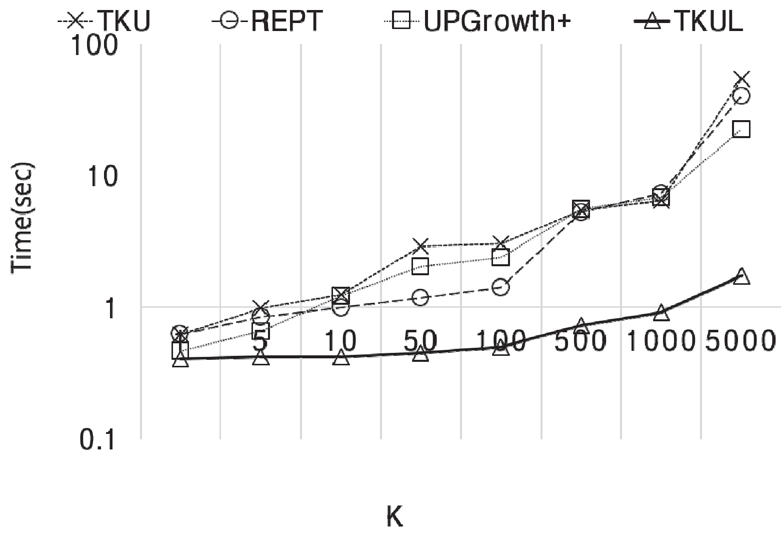


[그림 10] Accidents 데이터에 대한 실행시간

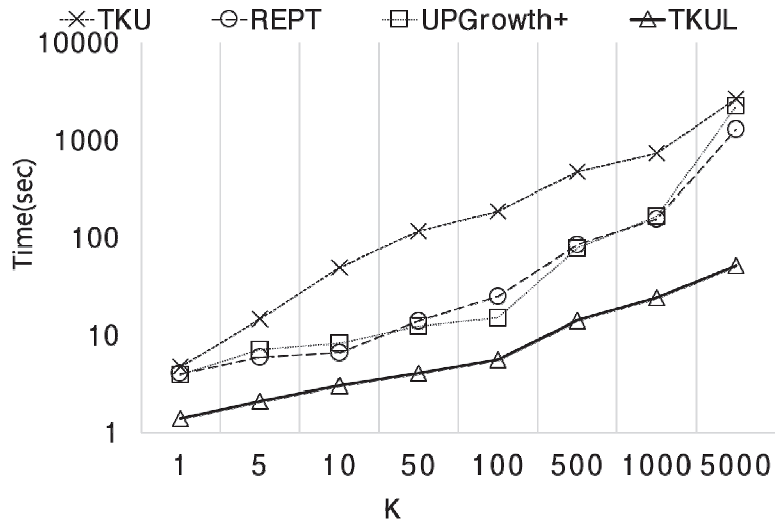


[그림 11] Chess 데이터에 대한 실행시간

[그림 12](REPT의 N = 100)와 [그림 13](REPT의 N = 1000)은 각각 생성 데이터 T10I4D100K와 T40I10D100K에 대한 실행시간 실험 결과를 보여준다. 두 실험에서 모두 TKUL-Miner 알고리즘의 성능이 다른 알고리즘들을 개선시켰다. [그림 12]는 비교적 작은 데이터 집합에 대한 실험 결과이기 때문에 k가 작을 때는 실행시간이 비슷하게 소요되나 k가 커짐에 따라 그 차이가 급격하게 나타난다. 그리고 [그림 13]에서는 TKUL-Miner가 실행시간의 차이가 더욱 커져 k가 1일 때는 약 10배 빠르고 k가 커질수록 [그림 12]보다 더 큰 차이로 개선시켰음을 보인다.



[그림 12] T10I4D100K 데이터에 대한 실행시간



[그림 13] T40I10D100K 데이터에 대한 실행시간

위 실험들을 통해 제안 알고리즘 TKUL-Miner가 대부분의 경우 다른 최신 알고리즘들과 비교하여 가장 빠른 실행시간을 소요한다는 것을 알 수 있다. 그리고 TKUL-Miner의 속도 개선의 정도는 데이터가 조밀한 분포를 가지면서 평균 트랜잭션의 길이가 긴 경우 더 크다.

2. 메모리 사용량 비교

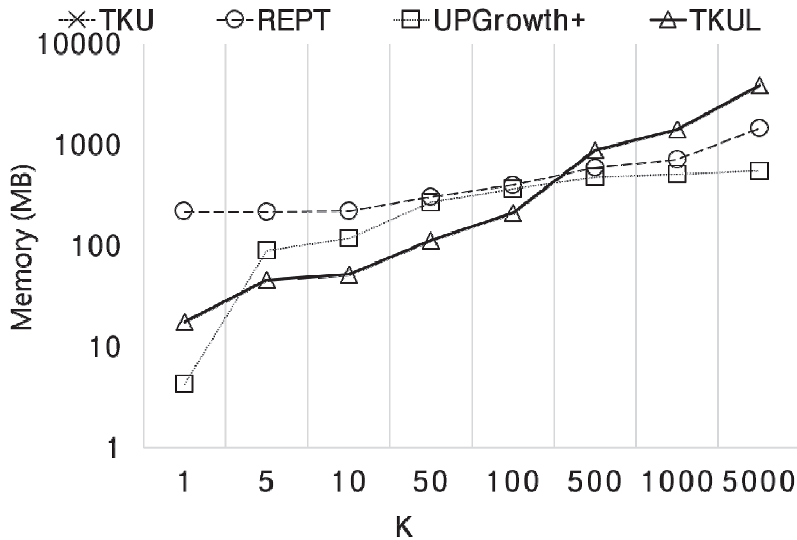
제안 알고리즘의 메모리 사용량을 평가하기 위해 실 환경 데이터와 생성 데이터에 대해 기존 알고리즘들과의 프로그램을 실행하는 동안 최대 메모리 사용량의 측정값을 보인다. 이 실험에서 REPT 알고리즘의 N 값은 실행시간 실험과 동일하게 설정했고 결과는 로그 눈금 간격으로 메가바이트(Megabyte) 단위를 표현하였다. 알고리즘이 메모리의 20,000 메가바이트 이상을 소요하는 경우 프로그램을 종료하였다.

[그림 14]는 희소한 데이터 Chain에 대한 메모리 사용량 측정 결과로 k가 1일 때는 최적의 유틸리티로 실행한 UP-Growth^{+} 알고리즘의 메모리 사용이 가장 효율적이나 k가 5일 때부터 100일 때 까지는 제안 알고리즘 TKUL-Miner의 메모리 사용이 가장 효율적이다. 그러나 k가 500 이상이 되면 제안 알고리즘의 메모리 사용량은 급격히 증가하게 된다. 이러한 결과를 보이는 이유는 제안 알고리즘이 후보를 생성하지 않아 상당히 메모리 효율이 높지만, 희소한 데이터의 경우 k가 커지게 되면 최소 유틸리티가 상당히 작게 설정되기 때문에 데이터에 대한 정보의 압축률이 상당히 높은 트리 구조 보다 메모리 효율이 떨어지게 된다.

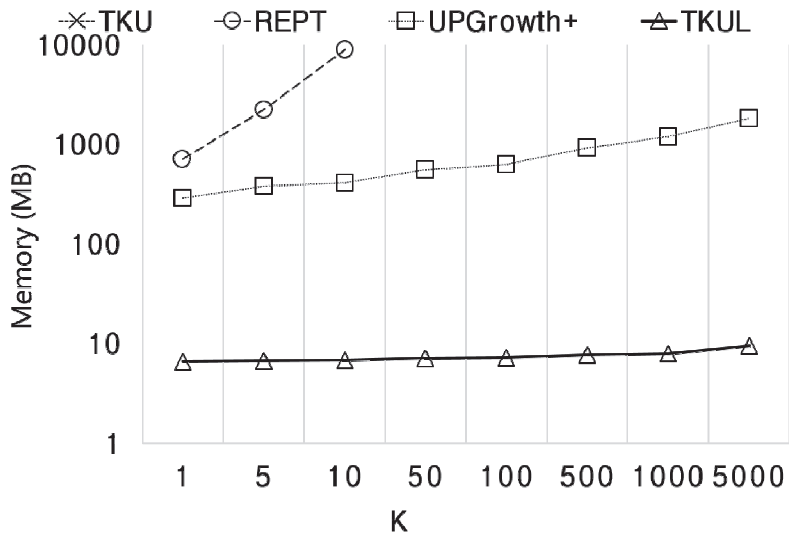
반면 조밀한 분포의 데이터를 갖는 Chess 데이터에 대한 실험 [그림

15]는 모든 k 값에 대해 제안 알고리즘 TKUL-Miner의 성능이 약 10배 이상 좋다는 것을 알 수 있다. 그리고 TKU 알고리즘은 k가 1일 때부터 너무나 많은 메모리를 소요하였고 그것보다는 개선된 REPT 알고리즘도 k 값이 10일 때 까지만 표현되어있다.

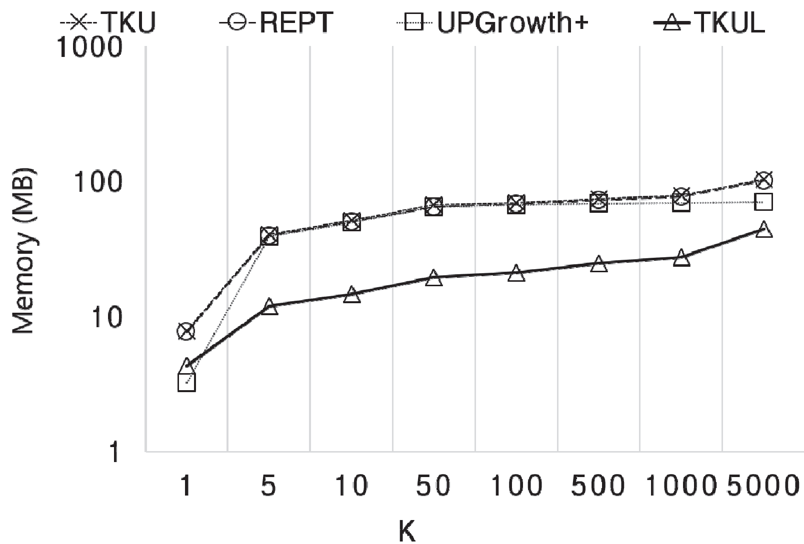
생성 데이터 T10I4D100K와 T40I10D100K에 대한 메모리 사용량 실험 결과는 [그림 16]과 [그림 17]에 나타나있다. 이 결과에 따르면, 상대적으로 크기가 작은 데이터에 대한 결과인 [그림 16]에서 k가 1일 때만 최적의 유틸리티로 마이닝을 수행한 UP-Growth+ 알고리즘의 메모리 사용량이 작고 나머지의 경우엔 제안 알고리즘인 TKUL-Miner 알고리즘의 메모리 사용량이 가장 작다. 그리고 대부분의 경우 가장 많은 후보를 생성하는 TKU 알고리즘의 메모리 사용이 비효율적이다.



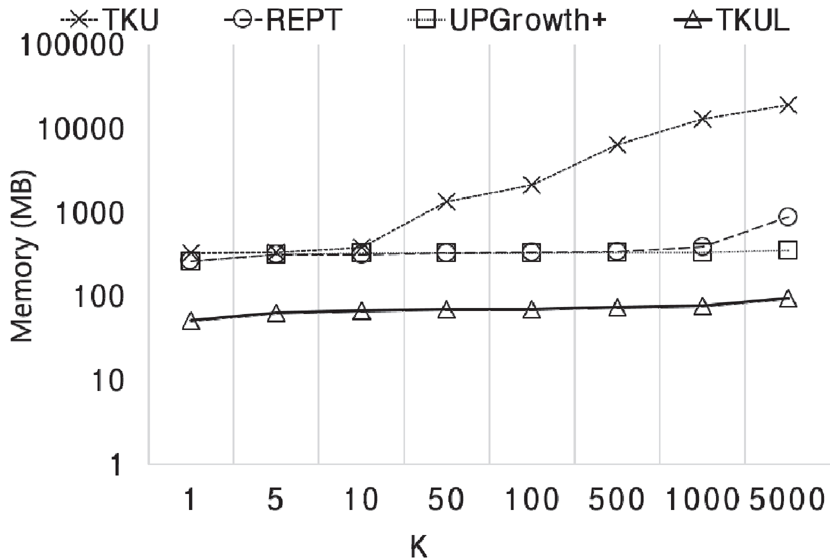
[그림 14] Chain 데이터에 대한 메모리 사용량



[그림 15] Chess 데이터에 대한 메모리 사용량



[그림 16] T10I4D100K 데이터에 대한 메모리 사용량



[그림 17] T40I10D100K 데이터에 대한 메모리 사용량

각 알고리즘의 메모리 사용량 비교를 통해 TKUL-Miner 알고리즘이 대부분의 경우 가장 효율적으로 메모리를 사용한다는 것을 알 수 있다. 특히 TKUL-Miner 알고리즘은 데이터가 조밀할수록 다른 알고리즘들과 비교해 메모리 효율이 더 높다. 이는 제안 알고리즘의 기반 구조인 유틸리티-리스트를 사용하기 때문에 후보 항목집합을 저장하지 않고 재귀적으로 트리를 구축하지 않기에 다른 알고리즘들 보다 메모리를 적게 사용하는 것이고, 그것이 데이터가 조밀할 때 더 큰 효과를 나타낸다. 그러나 제안 알고리즘은 데이터가 희소한 분포를 가질 경우 그 데이터의 트랜잭션의 개수에 따라 메모리 사용량이 영향을 받는다.

Ⅳ. 결론 및 향후 연구

본 논문은 top-k 높은 유틸리티 항목집합을 효율적으로 마이닝하기 위한 TKUL-Miner 알고리즘을 제안한다. TKUL-Miner 알고리즘은 유틸리티-리스트를 이용하여 기존의 top-k 높은 유틸리티 항목집합 마이닝 알고리즘에서는 필수적으로 수행해야했던 추가적인 데이터베이스 스캔을 하지 않는다. 제안 알고리즘은 최소 유틸리티를 빠르게 증가시키기 위해 항목집합 트리의 첫 번째 레벨에 대해 TWU 내림차순으로 탐색하는 전략을 이용한다. 또한 공통 유틸리티의 합 Sum_cutils와 남은 유틸리티인 rutil이 0인 항목집합의 유틸리티의 합을 이용하여 마이닝의 탐색 공간을 효과적으로 가지치기한다. TKUL-Miner 알고리즘은 실 환경 데이터와 생성 데이터에 대해 최신의 top-k 높은 유틸리티 항목집합 마이닝 알고리즘의 성능을 크게 개선하였다. 실험결과를 통해 대부분의 경우 실행시간과 메모리 사용을 줄였음을 보였다.

향후 연구로는 현재의 실행 속도를 유지하면서 보다 효율적인 메모리 사용을 할 수는 방법을 연구하는 것이 필요하다. 특히 본 연구의 제안 알고리즘은 데이터가 희소한 분포를 가진 경우에 대한 유틸리티-리스트 기반의 알고리즘이 트리 구조 만큼 자료를 압축하지 못하는데 두 알고리즘의 장점을 취해 어떤 경우에서도 최소한의 메모리를 사용할 수 있다면 실제 top-k 높은 유틸리티 항목집합 마이닝의 상용화에 커다란 이익을 줄 수 있을 것이다.

참 고 문 헌

[1] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules,” In *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Vol. 1215, pp. 487-499, 1994.

[2] 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.

[3] A.W.-C. Fu, R.W.-W. Kwong, and J. Tang, “Mining n-most interesting itemsets,” In *Proceeding of International Symposium on Methodologies for Intelligent Systems (ISMIS)*, Charlotte, Vol. 1932, pp. 59-67, 2000.

[4] J. Han, J. Wang, Y. Lu, and P. Tzvetkov, “Mining top-k frequent closed patterns without minimum support,” In *Proceedings of IEEE International Conference on Data Mining (ICDM)*, Maebashi, pp. 211-218, 2002.

[5] Y. Hirate, E. Iwahashi, and H. Yamana, “TF2P-Growth: an efficient algorithm for mining frequent patterns without any thresholds” , In *Proceedings of IEEE ICDM 2004 Workshop on Alternative Techniques for Data Mining and Knowledge Discovery*, Brighton, 2004.

[6] W. Wang, J. Yang, and P. Yu, "Efficient mining of weighted association rules (WAR)," In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, pp. 270-274, 2000.

[7] F. Tao, F. Murtagh, and M. Farid, "Weighted association rule mining using weighted support and significance framework," In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, pp. 661-666, 2003.

[8] Y. Liu, W. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," In *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, Vol. 3518, pp. 689-695, 2005.

[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, B.E. Shie, C.W. Wu, and P.S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 25, No. 8, pp. 1772-1786, 2013.

[11] M. Liu and J. Qu, "Mining high utility itemsets without candidate generation," In *Proceedings of the 21st ACM*

International Conference on Information and Knowledge Management, Maui, pp. 55-64, 2012.

[12] P. Fournier-Viger, C.W. Wu, S. Zida, and V.S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning," *Foundations of Intelligent Systems*, Springer, pp. 83-92, 2014.

[13] S. Lee and J.S. Park, "High utility itemset mining using transaction utility of itemsets," *KIPS Transactions on Software and Data Engineering*, to be published.

[14] M. Zihayat and A. An, "Mining top-k high utility patterns over data streams," *Information Sciences*, Vol. 285, pp. 138-161, 2014.

[15] T. Lu, Y. Liu, and L. Wang, "An algorithm of top-k high utility itemsets mining over data stream," *Journal of Software*, Vol. 9, No. 9, pp. 2342-2347, 2014.

[16] J. Yin, Z. Zheng, L. Cao, Y. Song, and W. Wei, "Efficiently mining top-k high utility sequential patterns," *IEEE 13th International Conference on Data Mining(ICDM)*, Dallas, pp. 1259-1264, 2013.

[17] C. Wu, B. Shie, V.S. Tseng, and P.S. Yu, "Mining top-k high utility itemsets," In *Proceedings of ACM SIGKDD 18th International Conferemce on Knowledge discovery and data mining*, New York, pp. 78-86, 2012.

[18] H. Ryang and U. Yun. "Top-k high utility pattern mining with effective threshold raising strategies," *Knowledge-Based Systems*, Vol. 76, pp. 109-126, 2015.

[19] J. Pisharath, Y. Liu, W.K. Liao, A. Choudhary, G. Memik, and J. Parhi, (2005). Numinebench version 2.0 dataset and technical report. Available at <http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html>. Accessed on June 2015.

[20] FIMI, (2003). Fimi: The frequent itemset mining dataset repository. Accessed on August 2015. <http://fimi.ua.ac.be/data/>.

ABSTRACT

Top-k High Utility Pattern Mining Based on Utility-List Structures

Serin Lee
Dept. of Computer Science
The Graduate School
Sungshin Women's University

Top−k high utility itemset mining refers to the discovery of top−k patterns using given user−specified value k by considering the utility of items in a transactional database. Since existing top−k high utility itemset mining algorithms are based on pattern−growth method, they search the patterns in two steps. Therefore, the generation of many candidates and additional database scan for calculating exact utilities are unavoidable. In this paper, we propose a new algorithm, TKUL−Miner, to mine top−k high utility itemsets efficiently. It utilizes a new utility−list structure which stores necessary information at each node on the search tree for mining the itemsets. The proposed algorithm has

a strategy using search order for specific region to raise the border minimum utility threshold rapidly. Moreover, two additional strategies for calculating smaller overestimated utilities are suggested to prune unpromising itemsets effectively. Experimental results on various datasets showed that the TKUL-Miner outperforms other recent algorithms both in runtime and memory efficiency.

Keywords: high utility itemset, top-k pattern mining, utility-list structure, data mining