PBL1. 친구추천

Similarity Join을 통한 효율적인 알고리즘 방안

1조 신민경 이정인 전성원 김덕영

Index

1: Problem Definition

Facebook의 데이터를 이용한 친구 추천 알고리즘 구현

Given Data

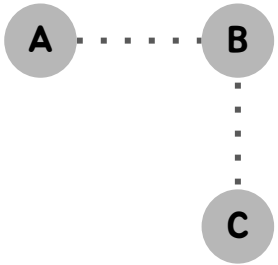
facebook의 친구 데이터, 친구관계를 이루는 (ID1, ID2) 쌍
Undirected Graph: ID1과 ID2는 '서로'친구관계이다.

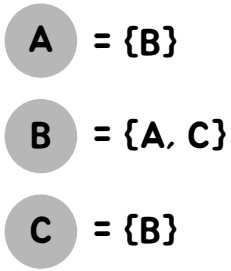
Goal

공통친구의 비율이 threshold τ (0.6, 0.7, 0.8, 0.9) 를 넘는 친구후보 쌍을 추천

=> 이때, **"아직 친구가 아닌"** 모든 후보를 추출하는 것이 핵심!

2: Similarity Join





GroupByKey()
FlatMap()

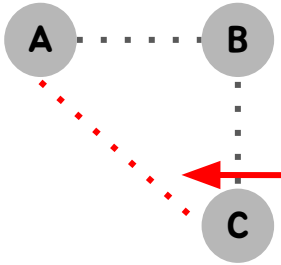
Item	Set ID
A	B
B	A, C
C	B

FlatMap()

Candidate Pair	Count
(A,C)	1

ReduceByKey()
Filter()

Candidate Pair	Count
(A,C)	1



2: Similarity Join

```
data = sc.parallelize(['A', ['C', 'D', 'E'], ['B', ['E', 'F'], ⋯])
x = data.flatMap(lambda l: [(x, (l[0], len(l[1]))) for x in (l[1])])
inverted_list = x.groupByKey().mapValues(list)
pair = inverted_list.flatMap(lambda l: [((l[1][i], l[1][j]), 1) for i in range(len(l[1])) for j in range(i+1, len(l[1]))])
overlap = pair.reduceByKey(lambda x, y: x+y)
answer =  overlap.filter(lambda t: t[1] >= a/((1+a)*(t[0][0][1]+t[0][1][1])))
                        .map(lambda t: (t[0][0][0], t[0][1][0]))
```

Limitation)
Inverted list를 계산 -> O(n^2) : 한 사람이 많은 친구를 갖는 경우에는 오랜 시간이 걸릴 수 있다.
Inverted list 내의 모든 원소에 대하여 후보 쌍을 생성하므로 중복이 많이 발생한다.

3: Algorithm Research

Efficient Similarity Joins for Near Duplicate Detection (ChuanXiao,et al In WWW, 2008)
Efficient Similarity Joins by Adaptive Prefix Filtering (JS Park, 2013)

〈Filtering Methods〉	〈Similarity Joins〉
Length Filtering	PPJoin
Prefix Filtering	PPJoin+
Positional Filtering	MPJoin
	APJoin

3: Algorithm Research - Length Filtering

: 아래 두가지 조건을 만족하지 않는 (x, y) 조인후보 쌍은 제외한다.

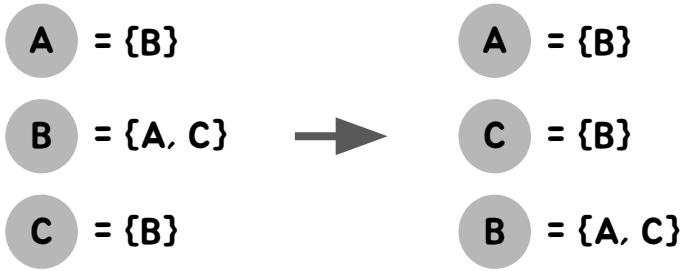
1) J(x, y) ≥ t ⇔ t*|x| ≤ |y|

$$O(x,y) \geq \alpha = \frac{t}{1+t}(|x| + |y|)$$

$$\alpha \leq |y|, \quad \frac{t}{1+t}(|x| + |y|) \leq |y|$$

$$t|x| \leq |y|$$

2) 집합의 크기가 작은 순으로
정렬하여 |y| ≤ |x|인 (x,y) 선택



3: Algorithm Research - Prefix Filtering

: 레코드 x, y에 대해, O(x, y) ≥ α를 만족할 때, (x의 p-prefix) ∩ (y의 p-prefix) ≥ 1이다.

U : 토큰의 전체집합

순서집합
O

레코드집합
R = {A,B,C}

O 에서 정의한
순서에 따라 정렬

A = {B}

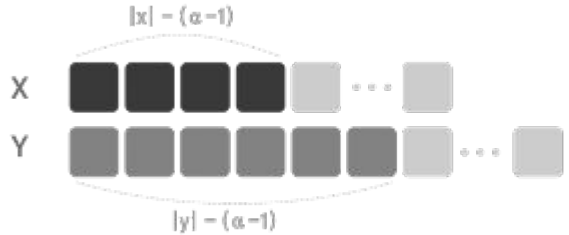
C = {B}

B = {A, C}

레코드 x의 p-prefix:
x의 토큰 중 앞에서부터 p개의 토큰

예시) 레코드 x의 3-prefix

x = {A, B, D, F}



레코드 x의 prefix 길이: |x| - α +1
레코드 y의 prefix 길이: |y| - α +1

-> 한 레코드의 prefix 길이는 다른 레코드 길이에 의해 결정된다.

∴ 최대 prefix 길이 = $|x| - \lceil t \cdot |x| \rceil + 1$

=> 조인후보 가능성이 없는 후보를 제외시켜 조인후보 쌍 수를 줄임

3: Algorithm Research - Positional Filtering

: Overlap의 최대크기 ≥ α 를 만족하지 못하는 레코드를 조인후보 쌍 생성 전에 제외한다.

레코드 x, y의 공통토큰이 각각 i번째, j번째 있다고 하자.

Overlap의 최대크기 = 현재 Overlap 크기 + min(len(x)-i, len(y)-j)

prefix filtering은 prefix 범위 내에서는
O(x,y) ≥ α를 만족하지 못하더라도 조인후보 쌍으로 선택한다.
-> 불필요한 조인후보 쌍이 많이 생길 수 있다.

∴ positional filtering 이용하면, 생성되는 조인후보의 쌍을 많이 줄일 수 있다.

3: Algorithm Research - Similarity Joins

PPJoin

-지난 수업시간에 배웠던 알고리즘
-단순 Similarity Join 알고리즘 + 세가지 Filtering 기법(Length + Prefix + Positional)

하지만 여전히 많은 조인후보 쌍을 만들어 조인후보 검증 시간이 많이 든다.

PPJoin+

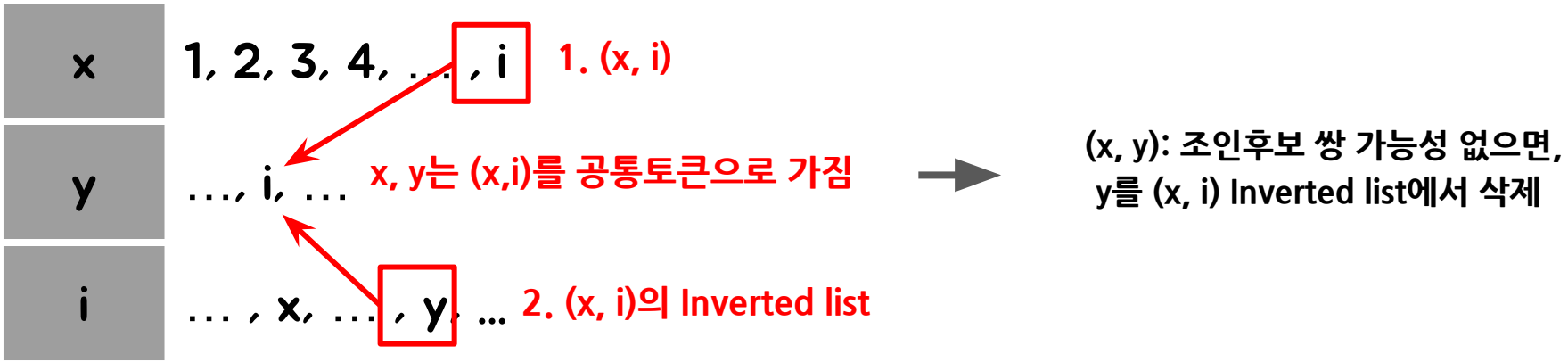
-PPJoin을 개선하기 위해 새로운 filtering 기법(suffix filtering) 사용

조인후보 쌍 수는 많이 줄였지만, 조인후보 쌍을 만들기 위한 비교연산은 PPJoin과 동일해
큰 성능개선은 하지 못했다.

3: Algorithm Research - Similarity Joins

동적 Prefix Filtering

1. 레코드 x의 i번째 토큰 (x, i) => (x, i) 는 x를 원소로 가진다.
2. (x, i)의 Inverted list의 토큰 중 하나를 y라고 할 때, x와 y는 (x, i)를 공통토큰으로 가진다.
3. 이 때 (x, y)가 조인후보 쌍이 될 가능성이 없으면, y를 (x, i) Inverted list에서 항목을 삭제



3: Algorithm Research - Similarity Joins

MPJoin

-Inverted list를 만들면서 y의 prefix 범위를 넘는 토큰을 Inverted list에서 삭제
-공통 토큰이 y의 prefix 범위 내에 있고, positional filtering을 만족하면 후보 쌍으로 생성

조인후보 쌍이 유의미하게 감소

APJoin

-MPJoin과 동일하게 동적 prefix filtering을 사용
-하지만 y뿐만 아니라 매번 달라지는 x의 prefix 범위도 고려

따라서 조인후보 쌍을 추출하기 위한 비교연산 감소

3: Algorithm Research - Similarity Joins

APJoin Pseudo-Code

```
Algorithm: APJoin(R, t)
1  S ← ∅;
2  I_w ← ∅ (1 ≤ w ≤ |U|);     // initialize inverted index
```

S[] = 조인결과 쌍을 저장
I[] = Inverted list

```
3  for each x ∈ R do
4    A ← empty map from record id to int;
5    max_probe_prefix ← |x| − ⌈t · |x|⌉ + 1;          // Eq. (5)
6    max_index_prefix ← |x| − ⌈2|x| · t/(t+1)⌉ + 1; // Eq. (6)
7    for i = 1 to max_probe_prefix do
8      α = ⌈(|x| + |x| − i + 1) · t/(t+1)⌉;     // |y| = |x| − i + 1
9      prefix_x[i] = |x| − α + 1;                    // Eq. (3)
10     prefix_y[i] = (|x| − i + 1) − α + 1;          // Eq. (4)
```

A[] = y의 Overlap score를 저장
각 레코드 별로 α와 prefix 범위를 계산해서 저장

```
13     for each (y, j) ∈ I_w do          // 레코드 y의 토큰 (y, j)
14       if |y| < t · |x|                // 현재 노드 이후 삭제
15         (y, j)와 연결된 노드들 삭제;
16       else if j > prefix_y[|x| − |y| + 1]
17         (y, j) 삭제;                  // 현재 노드 삭제
18       else if i > prefix_x[|x| − |y| + 1]
19         continue;                     // 다음 노드로 이동
20       else
21         A[y] = A[y] + 1;              // candidate generation
```

동적 Prefix filtering
1) 레코드 (y,j)가 length filtering을 만족하지 않는 경우, 토큰 (y,j)를 Inverted index에서 삭제
2) j가 y의 prefix 범위를 벗어나면 토큰 (y,j)를 Inverted index에서 삭제
3) i가 x의 prefix 범위를 벗어나면 다음 토큰으로 이동
4) 위 조건들에 해당하지 않는 경우, 조인후보 쌍이 될 수 있으므로 A리스트의 값을 증가

```
22   for i = 1 to max_index_prefix do
23     w ← x[i];
24     I_w = I_w ∪ {(x, i)};      // 토큰 (x, i)를 inverted index에
25   VerifyZip(x, A, α);          // verification phase
26 return S
```

Inverted list를 생성하고, 조인후보 쌍을 검증하고 출력

3: Algorithm Research - similarity Joins

논문 내 4가지 Join들의 성능비교

	|Comp|	1) |Cand|	|Join|	Time(sec)
PPJoin 2)	4,132,651	1,356,337	3,683	2.106
PPJoin+	4,132,651	11,905	3,683	1.810
MPJoin	2,200,083	1,356,628	3,683	0.978
APJoin	1,378,845	1,357,015	3,683	0.668

1) PPJoin vs. PPJoin+: 비교연산 수는 동일, 추출된 조인후보 쌍은 큰 차이
2) PPJoin vs. (MPJoin & APJoin): 비교연산 수는 큰 차이, 추출된 조인후보 쌍은 차이 없음

4: Codes & Results

친구리스트 작성/정렬

In [1]:
```
import findspark
findspark.init()
```

In [2]:
```
from pyspark import SparkContext
from pyspark.sql import SQLContext, SparkSession

sc = SparkContext("local", "pbl-1")
spark = SparkSession(sc)
```

In [3]:
```
data = sc.textFile("facebook_combined.txt")

parse_data = data.flatMap(lambda line: line.split('\n')) \
            .map(lambda l:l.split(' ')).map(lambda l: [(int(l[0]),int(l[1])),(int(l[1]),int(l[0]))]).flatMap(lambda t:
```

In [4]:
```
group = parse_data.groupByKey().mapValues(list)
new_group = group.flatMap(lambda l: [(l[0],(len(l[1]), sorted(l[1])))])
#print(new_group.collect())
```

In [5]:
```
order = new_group.sortBy(lambda v: v[1])
R = order.mapValues(lambda v: v[1]).collect()
#print(R)
```

4: Codes & Results

변수 선언/α 값·prefix 범위 계산

Algorithm: APJoin(R, t)

```
1  S ← ∅;
2  I_w ← ∅ (1 ≤ w ≤ |U|);    // initialize inverted index
3  for each x ∈ R do
4    A ← empty map from record id to int;
5    max_probe_prefix ← |x| − ⌈t · |x|⌉ + 1;          // Eq. (5)
6    max_index_prefix ← |x| − ⌈2|x| · t/(t+1)⌉ + 1;  // Eq. (6)
7    for i = 1 to max_probe_prefix do
8      α = ⌈(|x| + |x| − i + 1) · t/(t+1)⌉;    // |y| = |x| − i + 1
9      prefix_x[i] = |x| − α + 1;                // Eq. (3)
10     prefix_y[i] = (|x| − i + 1) − α + 1;      // Eq. (4)
11   for i = 1 to max_probe_prefix do
12     w ← x[i];                    // 레코드 x의 토큰 (x, i)
13     for each (y, j) ∈ I_w do     // 레코드 y의 토큰 (y, j)
14       if |y| < t · |x|           // 현재 노드 이후 삭제
15         (y, j)와 연결된 노드들 삭제;
16       else if j > prefix_y[|x| − |y| + 1]
17         (y, j) 삭제;             // 현재 노드 삭제
18       else if i > prefix_x[|x| − |y| + 1]
19         continue;                // 다음 노드로 이동
20       else
21         A[y] = A[y] + 1;         // candidate generation
22   for i = 1 to max_index_prefix do
23     w ← x[i];
24     I_w = I_w ∪ {(x, i)};        // 토큰 (x, i)를 inverted index에
25   VerifyZip(x, A, α);            // verification phase
26 return S
```

```python
r_set = [0.6, 0.7, 0.8, 0.9]
set_idx = {id:i for i,(id,x) in enumerate(R)}

times_ = []
length_ = []

for r in r_set:
    print("# when r = ",r)
    start = time.time()

    S = []
    I = [[] for _ in range(len(R))] # inverted list
```

```python
for (x_id, x) in R:
    max_probe_prefix = len(x) - math.ceil(r*len(x)) + 1
    max_index_prefix = len(x) - math.ceil(2*len(x)*r/(r+1)) + 1
    prefix_x = [0]
    prefix_y = [0]
    A = [0 for _ in range(len(R))] # overlap score for y

    for i in range(1,max_probe_prefix+1):
        a = math.ceil((2*len(x)-i+1)*r/(r+1))
        prefix_x.append(len(x)-a+1)
        prefix_y.append(len(x)-i+1-a+1)
```

4: Codes & Results

동적 Prefix Filtering/Inverted list 작성

Algorithm: APJoin(R, t)

```
1  S ← ∅;
2  I_w ← ∅ (1 ≤ w ≤ |U|);     // initialize inverted index
3  for each x ∈ R do
4    A ← empty map from record id to int;
5    max_probe_prefix ← |x| − ⌈t · |x|⌉ + 1;          // Eq. (5)
6    max_index_prefix ← |x| − ⌈2|x| · t/(t+1)⌉ + 1; // Eq. (6)
7    for i = 1 to max_probe_prefix do
8      α = ⌈(|x| + |x| − i + 1) · t/(t+1)⌉;     // |y| = |x| − i + 1
9      prefix_x[i] = |x| − α + 1;                  // Eq. (3)
10     prefix_y[i] = (|x| − i + 1) − α + 1;        // Eq. (4)
11   for i = 1 to max_probe_prefix do
12     w ← x[i];                          // 레코드 x의 토큰 (x, i)
13     for each (y, j) ∈ I_w do           // 레코드 y의 토큰 (y, j)
14       if |y| < t · |x|                 // 현재 노드 이후 삭제
15         (y, j)와 연결된 노드들 삭제;
16       else if j > prefix_y[|x| − |y| + 1]
17         (y, j) 삭제;                   // 현재 노드 삭제
18       else if i > prefix_x[|x| − |y| + 1]
19         continue;                      // 다음 노드로 이동
20       else
21         A[y] = A[y] + 1;               // candidate generation
22   for i = 1 to max_index_prefix do
23     w ← x[i];
24     I_w = I_w ∪ {(x, i)};      // 토큰 (x, i)를 inverted index에
25   VerifyZip(x, A, α);                  // verification phase
26 return S
```

```python
for i in range(1,max_probe_prefix+1):
    w = x[i-1]
    w_idx = set_idx[w]

    for ((y_id, y),j) in I[w_idx]:
        if len(y) < r*len(x):
            I[w_idx].remove(((y_id,y),j))
        elif j > prefix_y[len(x)-len(y)+1]:
            I[w_idx].remove(((y_id,y),j))
        elif i > prefix_x[len(x)-len(y)+1]: continue
        else:
            A[set_idx[y_id]] = A[set_idx[y_id]] + 1 # 후보 확정
```

```python
for i in range(1,max_index_prefix+1): # x의 원소에 대한 inverted list 추가
    w = x[i-1]
    w_idx = set_idx[w]
    I[w_idx].append(((x_id,x),i))
```

4: Codes & Results

조인 후보 쌍 검증

Algorithm: APJoin(R, t)

```
1  S ← ∅;
2  I_w ← ∅ (1 ≤ w ≤ |U|);     // initialize inverted index
3  for each x ∈ R do
4    A ← empty map from record id to int;
5    max_probe_prefix ← |x| − ⌈t · |x|⌉ + 1;          // Eq. (5)
6    max_index_prefix ← |x| − ⌈2|x| · t/(t+1)⌉ + 1; // Eq. (6)
7    for i = 1 to max_probe_prefix do
8      α = ⌈(|x| + |x| − i + 1) · t/(t+1)⌉;     // |y| = |x| − i + 1
9      prefix_x[i] = |x| − α + 1;                  // Eq. (3)
10     prefix_y[i] = (|x| − i + 1) − α + 1;       // Eq. (4)
11   for i = 1 to max_probe_prefix do
12     w ← x[i];                          // 레코드 x의 토큰 (x, i)
13     for each (y, j) ∈ I_w do           // 레코드 y의 토큰 (y, j)
14       if |y| < t · |x|                 // 현재 노드 이후 삭제
15         (y, j)와 연결된 노드들 삭제;
16       else if j > prefix_y[|x| − |y| + 1]
17         (y, j) 삭제;                   // 현재 노드 삭제
18       else if i > prefix_x[|x| − |y| + 1]
19         continue;                      // 다음 노드로 이동
20       else
21         A[y] = A[y] + 1;               // candidate generation
22   for i = 1 to max_index_prefix do
23     w ← x[i];
24     I_w = I_w ∪ {(x, i)};              // 토큰 (x, i)를 inverted index에
25   VerifyZip(x, A, α);                  // verification phase
26 return S
```

```python
for i in range(len(R)): # 후보 검증 후 S에 추가
    (y_id, y) = R[i]

    if A[i] == 0: continue # prefix 안에 공통 토큰이 하나도 없다면 제외

    a = math.ceil((len(x)+len(y))*r/(r+1))

    (p,q,cnt) = (0,0,0)
```

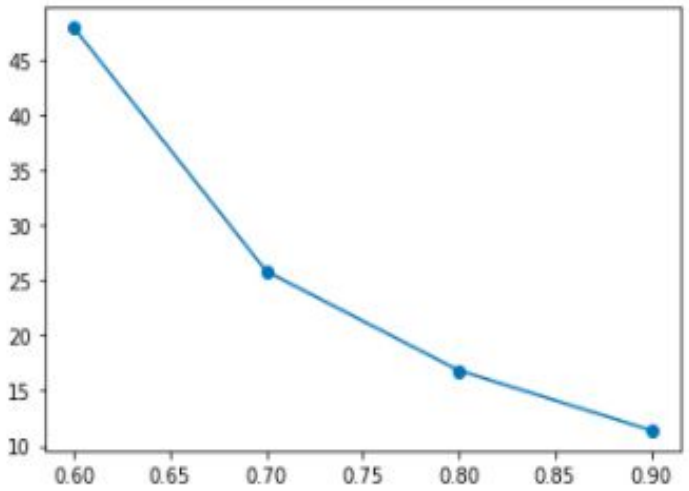
```python
while(p<len(x) and q<len(y)):# 공통 토큰 개수 카운트
    if x[p] == y[q]:
        cnt = cnt+1
        p = p+1
        q = q+1
    elif cnt+min(len(x)-p-1,len(y)-q-1) < a: break # positional filtering
    elif x[p] < y[q]: p = p+1
    else: q = q+1
if cnt >= a and y_id not in x: # x와 y가 이미 친구인 경우는 제외
    S.append((x_id, y_id))
```

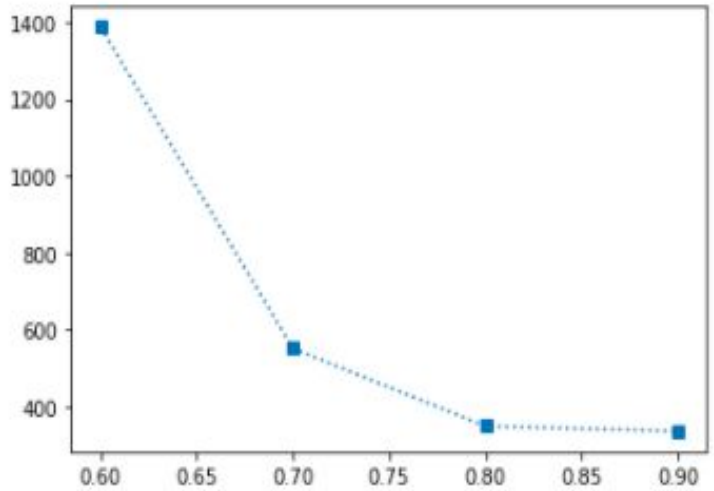
4: Codes & Results

실행시간/결과 조인 수 비교

```
import matplotlib.pyplot as plt
plt.plot( r_set, times_, linestyle='-', marker='o')
plt.show()

plt.plot( r_set, length_, linestyle=':', marker='s')
plt.show
```





=> threshold가 커질 수록, 실행시간/조인결과 쌍의 개수 모두 감소

5: Additional Research

1. Adamic Adar(아담 아다르) 알고리즘:
공유하고 있는 이웃을 기반으로 각 노드의 근접성을 계산
-> 즉, 노드 A와 B가 공유하고 있는 친구가 많더라도 그 친구들의 거리가 멀면 A와 B가 서로 친구가 될 가능성이 적다고 평가

2. Common Neighbors 알고리즘:
연결되어 있지 않은 두 노드들 간에 공통의 노드들이 공유된다면, 이 둘은 이후에 연결될 가능성이 높다는 원리를 이용

3. Friend of Friend Algorithm(FOAF) 알고리즘:
연결길이 l을 사용해서 한 개체와 거리가 l인 다른 개체를 그룹으로 형성
-> 즉, 개체들의 연결망을 형성하여 친구를 추천

4. SimRank 알고리즘:
“비슷한 사람에 의해서 가리켜지면, 비슷한 사람일 것이다”라는 가정에 기반한 Node Similarity 알고리즘.
*이를 계산하는 과정이 다음주에 배우게 될 PageRank와 매우 유사

감사합니다