Matching Users With Common Attributes

Deniz Cakiroglu

1 Introduction

This paper describes an algorithm for matching users that have given set of attributes. It matches the users based on how many attributes they have in common. The attributes can be weighted. This algorithm was developed for matching users by their most listened artists and songs retrieved from Spotify. The implementation of this algorithm can be found here.

2 Queueing Preference Changes

Prefs are received in a list for each user. For each element in the list, that pref's existing weight for given user is checked. If that pref exists and the weight is different, that pref is queued. If it doesn't exist, the pref is also queued.

Assume i, p, w stands for user id, pref id and weight respectively.

$A = \{$ most recent user prefs $\}$
with A_i representing the preference list of a given user
with A_{i_p} representing the specific preference weight in the list of a given user
This list is received externally, and used to populate Q, which after is used to update P.

$Q = \{$ prefs to be updated in the algorithm database $\}$
Each element in this set is a list containing $\{i, p, w\}$

$P = \{$ existing prefs for all users in database $\}$
with P_p representing the user list for a given preference
with P_{p_i} representing the weight of this preference for specific user

The operation to populate Q can be defined as:
$(\forall_{i,p} \in (A \cup P))(A_{i_p} \cap \overline{P}_{p_i}) = Q$

Implementation of this operation can be found in the following function:
`https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L46`

3 Applying Queued Changes to the graph

To apply the changes, we first subtract the existing weight for each given queued change. Otherwise, we would be adding new weight of a changed pref on top of the old one. After, subtraction is done, we add the changed pref.

$G = \{$ Graph containing total weight of all common prefs between each user with common prefs $\}$
G_{i_j}: total weight of common prefs of user i and j

The set of edges after subtraction
$(\forall_q \in Q, \forall_j \in (G_i \cap P_{q.i}))(G_{q.i_j} : G_{q.i_j} - P_{q.i}, G_{j_{q.i}} : G_{j_{q.i}} - P_{q.i}))$
Re-adding the new weights
$(\forall_q \in Q, \forall_j \in (G_i \cap P_{q.i}))(G_{q.i_j} : G_{q.i_j} + q.e, G_{j_{q.i}} : G_{j_{q.i}} + q.w))$

Implementation of this operation can be found in the following function:
https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L234

4 Getting Matches

In theory, getting a match should be fairly simple.
$\forall_i(max(G_i))$ gives j which is the match.

Assume we store already matched users in M and don't want duplicate matches.
$M = \{$ previously matched users $\}$
where M_i gives the list of users it has been matched
where M_{i_j} exists, there has been a match between the two users

Then
$\forall_i(max(G_i \cap \overline{M}_i))$ to exclude the already matched users.

What if there are no matches left? In other words, what if a user has previously matched with all it's neighbors in the graph or has no neighbors? Or maybe all their neighbors have already matched with someone in this iteration of the algorithm. The neighbors only consist of other users that the given user has common attributes with.
$L = \{$users that couldn't match their neighbors, i.e. $G_i \cap \overline{M}_i = \emptyset\}$

For any given user in L, any other user that hasn't been matched with the given user must be non neighboring.

We match people in L with each other.
$\{\forall_{i,j} \in G(i \neq j, \nexists M_{i_j})\}$

```
L_matched = Set()
for u1 in L:
    if u1 in L_matched:
        continue
    found = False
    for u2 in l:
        if u1 != u2 and u2 not in L_matched and not M.u1.u2:
            L_matched.add(u1)
            L_matched.add(u2)
            found = True
            M.u1.u2 = True
            M.u2.u1 = True
            break
    if not found:
        LL.add(u1)
```

We can't guarantee everyone in L gets a match. It is possible that for a given user in L, all the other users have gotten a match in this iteration or given user has matched them already before.
$LL = \{$ user(s) that couldn't match with other users in L $\}$
We match the user(s) in LL with someone random among all users. Which means, it is possible for some users to get more than 1 match a day.
$\{\forall_i \in LL, \forall_j \in G, \nexists M_{i_j}, \nexists M_{j_i}\}$

What if multiple users in LL end up matching the same user?
To prevent this, we hard code a match count.
We allow the user(s) in LL to match with their selected user only if the selected person had not been matched more than twice.

Why twice?
Because if only 1 user remains in LL with no matches due to all possible matches having matched once this iteration, it wouldn't be able to find a match.
This means, up to 2 matches a day are possible in certain situations for a few lucky users.

```
LL_matched = Set()
for u1 in LL:
    if u1 in LL_matched:
        continue

    for u2, _ in G:
        m1 = matched_today.has(u2) ? 1 : 0
        m2 = L_matched.has(u2) ? 1 : 0
        m3 = LL_matched.has(u2) ? 1 : 0
        if M.u1.u2 or (m1+m2+m3 > 2):
            continue
        M.u1.u2 = True
        M.u2.u1 = True
        LL_matched.add(u1)
        LL_matched.add(u2)
```

The complete implementation of step 4 can be found in this function:
https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L55

5 Additional Notes

In the example implementation, constraints such as common languages and active users are also included. Adding any constraint set is pretty simple.
For example:
$LA = \{$ users with common languages $\}$
where LA_i is the set of other users that user i can speak with
where LA_{i_j} means user i and j understand each other.

So, $\forall_i(\max(G_i \cap \nexists M_i))$ becomes $\forall_i(\max(G_i \cap \nexists M_i \cap LA_i))$.
And this repeats in every step that involves matching. Any other constraint can be added this way. At the time this paper is written, the last added constraint was the active user set.