FOAF-Based Caching in Friend Recommendation

Final Project Report

Abdullah Caglar Oksuz
Bilkent University
caglar.oksuz@bilkent.edu.tr

Cihan Eryonucu
Bilkent University
cihan.eryonucu@bilkent.edu.tr

ABSTRACT

Social networks are composed of various organizations, groups and individuals. These elements are interconnected with high-complexity. Friend recommendation is an important feature in these highly-complex environments to establish new links between unconnected individuals. This feature is expected to be provided to users with high precision, so that they won't spend thousands of hours looking all the users of social networks to find their actual friends. Hereby, we propose a new algorithm of friend recommendation to find friends based on who we added "recently". By using this algorithm, we expect to find potential friends from similar communities much easier. In our algorithm, we take recently added n users, find their pairwise common friend vectors and rank these common friends based on how many pair vectors they appear in.So, appearing in most pairs increases the ranking of a user. We plan to evaluate the efficiency of algorithm, using R-Precision and we expect our algorithm to be more efficient than friend recommendation using number of common friends approach.

Keywords

Friend Recommendation, Caching, Ranking, Social Networks, Social Media Search

1. INTRODUCTION

Friend recommendation is an important feature of many social networks like Facebook, Twitter etc. These social networks use variety of algorithms to recommend possible friends to their users. An example of such algorithm is Friend-of-a-friend (FOAF) algorithm which evaluates the interaction of not only you with other people but also of your social communities. However, we still may not be able to find intended friends in similar social communities (school, club friends, religious communities etc.) using regular friend recommendation systems in social networks.

FOAF Based Caching is prioritizing recently added friends for friend recommendation. Therefore, detecting target people in similar social communities with recently added friends is much easier. For example, we want to add our primary school friends who are possibly friends with each other. When we add a couple of them using regular search, we expect to see more people that may be related with the ones we recently added. There is a high probability of detecting possible primary school friends from the common friends of recently added ones.

The rest of the paper will be organized as follows: In Section 2, we will describe our methodology steps which are, calculating pairwise similarity vectors using the current users in cache, updating cache and P matrix (Pairwise Common Friends Matrix), calculation of ranks using P matrix and adding new user. Finally In Section 3, we will explain our expected results and acceptance rates.

2. METHODOLOGY

We will hold a pair-wise common friends matrix, P matrix, for our operations. We will mainly use this matrix for our friend recommendation operation.P is 3-Dimensional matrix with size of $k \times k \times n$ which k is a cache size and n is total number of users. P matrix's first and second, x and y, dimensions indicate the friend ID's. Third dimension, z, will be common friends of the x and y. p_{ij} is common friend list of i and j. Top row of the P matrix is least recently added friend, therefore first element to be removed from cache. Last row is most recently added friend. An example of P matrix is below with cache size 6.

$$\begin{Vmatrix} x & 12 & 13 & 14 & 15 & 16 \\ x & x & 23 & 24 & 25 & 26 \\ x & x & x & 34 & 35 & 36 \\ x & x & x & x & 45 & 46 \\ x & x & x & x & x & 56 \\ x & x & x & x & x & x \end{Vmatrix}$$

For example, entry 12 has common friends of user 1 and 2. There can be at most n friends, n is number of the all users, thus P has size of $k \times k \times n$. We need only upper half of the P matrix since 12 and 21 is the same thing. In addition, calculation of 11 redundant obviously therefore we do not need to calculate it.

2.1 Dataset

In our project, we used Facebook Friendships Network of Koblenz Network Collection from University of Koblenz-Landau as a dataset [2]. Facebook Friendship Network is undirected and unweighted. It has 63,731 vertices (users) and 817,035 edges (friendships). Since no additional features are provided, we only relied on the friendship status of users. For creating realistic test scenarios, we assumed that the users we want to find forms a connected sub-graph among themselves. Therefore, we modified the network to satisfy our condition. For realism and testing purposes, we extracted a network of 25 facebook friends from similar specific community and added them into our dataset. Later on, these users are marked as the users we need to find and some of them are added into cache. Using the pre-knowledge

obtained from cache we tested our algorithm.

2.2 Link Generation

In order to work on as many test cases as we want and to make our experiment more scientific through random assignment process, we devised a strategy that will add additional friends to our self-extracted network. First of all, we created an upper triangular matrix that keeps the friendship statuses of self-extracted users. This matrix uses the same structure of Koblenz Friendship Network for storage. Only, users with lower IDs can hold the friendship status with upper ID users. Thanks to this structure, upper ID users don't need to store friendship status with lower ID users which reduces the amount of space used into half. Secondly, these users are assigned with random IDs within the network which is from 1 to 63731. Then using the upper triangular matrix from top to bottom, we generated new friendship links to be added into the friendship statuses Koblenz Network. For existing users, adding new links can be thought as overriding. However, deletion of previous friendship links are prevented. By this way, we guaranteed that self-extracted users will have different additional friends through assignment on random pre-existing users. Finally, modified input file having the friendship links of both Koblenz and self-extracted network is named "merger" to be used in test cases and can be modified randomly in each iteration. A final remark to point out is that we can change the friendship links of self-extracted network by changing the values of link matrix in linkGenerator() method.

2.3 Algorithm

Our proposed algorithm firstly generates the P matrix according to two things. First one is cache size k. Second one is recently added k friends. Using these P matrix is constructed. Finding common friends between two users follows the below CommonFriends method which takes friend list of users and returns a common friends list.

CommonFriends(F_i,F_j)

```
1: n_i ← F_i.length
2: n_j ← F_j.length
3: commonFriends ← ∅
4: for i=1 to n_i do
5:    for j=1 to n_j do
6:       if F_i[i] == F_j[j] then
7:          commonFriends.add(F_i[i])
8:          break
9:       end if
10:   end for
11: end for
12: return commonFriends
```

We needed to have ranking method to rank the users according to their number of common friends. Recommendation is based on this ranking. Rank method counts the appearances of users and stores them in counts array. For example, if user i is in the P matrix, Rank method increments the $counts[i]$ value by one. For input parameters, P is the famous P matrix. h is the number of top ranked users and finally $CurrentFriends$ is friends of actor.

Rank(P,h,$CurrentFriends$)

```
1: k ← P.length
2: counts ← ∅
3: for i=1 to k do
4:    for j=i+1 to k do
5:       l ← p_ij.length
6:       for x=1 to l do
7:          counts[P[i, j, x]] + +
8:       end for
9:    end for
10: end for
11: counts = counts/CurrentFriends {/ is set difference}
12: toph ← Max(counts, h)
13: return toph
```

We will explain this pseudo-code for better understanding of our approach. 1st line obtains the cache size, k. 2nd line initializes the counts, initially all users have a value 0. Lines 3-10, basically counts the appearances of users in P matrix by going over the all users of P matrix. Line 11 excludes the current friends from ranking so that Rank method will not recommend already added friends. Finally, line 12 picks the top h users and returns them.

When user added another user as a friend, let's say he/she added friend whose id is 7, we will alter the P matrix. That is because our algorithm is cache based meaning that it will use only k recently added friends. Therefore, least recently added friend will leave the P matrix and new user will be added. Resulting P matrix is below after the insertion of user 7.

$$\begin{Vmatrix} x & 23 & 24 & 25 & 26 & 27 \\ x & x & 34 & 35 & 36 & 37 \\ x & x & x & 45 & 46 & 47 \\ x & x & x & x & 56 & 57 \\ x & x & x & x & x & 67 \\ x & x & x & x & x & x \end{Vmatrix}$$

Note that after the updating the P matrix we are at the initial stage of our algorithm. This procedure will repeat itself until the user terminates. General algorithm is below. It will take the friendship graph, G, marked users that we are looking for, V', cache size, k, number of top ranked users, h, number of users that are market r and initial friends, $friends$. r value is number of iterations for friend recommendations therefore directly involves the r-precision. Note that we can access vertices of graph by $G.V$ and edges of graph $G.E$.

FOAF-Cache(G,V',k,h,r,$friends$)

```
1: n ← G.V.length
2: P ← empty k × k × n sized matrix
3: i ← 0
4: CurrentFriends ← ∅
5: while i < k do
6:    P.insert(friends[i])
7:    CurrentFriends.add(friends[i])
8:    i = i + 1
9: end while
10: for i=1 to k do
```

```
11:    j ← i + 1
12:    while j < k do
13:      p_ij ← CommonFriends(G.E[i], G.E[j])
14:    end while
15:  end for
16:  for i=1 to r do
17:    rankingArray ← Rank(P, h, CurrentFriends)
18:    P.remove()
19:    selected ← false
20:    for u ∈ rankingArray do
21:      if u ∈ V' then
22:        CurrentFriends.add(u)
23:        P.insert(u)
24:        selected ← true
25:      end if
26:    end for
27:    if !selected then
28:      pick a random user u from rankingArray
29:      CurrentFriends.add(u)
30:      P.insert(u)
31:    end if
32:  end for
```

Lines 5-10 initially fills the P matrix with initial friends. For recommendation to start it will need to fill the P matrix. This process can be done with initial given friends. There will be no recommendations until P matrix is is filled. $insert(friend)$ method always inserts to the last row of P matrix. Lines 10-15 is calculating the common number of friends in all pairs of friend in the cache, P. Line 17 ranks the users. Ranking is basically appearance of the user in p_{ij}'s. Rank method is also given above with its structure and input/output parameters. $rankingArray$ variable contains the top ranked h values. Line 18 removes the first row, least recently added friend, from the cache, P. Lines 20-26 will look for the marked friends that are we looking for. If we find such a friend we will add it to the cache. If we do not find a such friend we will add the top ranked friend as shown in lines 27-31. This procedure, lines 16-32, will repeat r times.

2.4 Expectation Criteria

For testing the results, we set a user from network as our own user u. Before testing, we had the pre-determined list of n users to add which are assumed to be from the same community. Then, our own user regularly adds a couple of users from the list. These recently added users then recorded into the cache. After that, we expect to see recommendations from our algorithm. We define successful recommendation as among the top x recommended users at least one of them must be in predetermined list. Then we will look at the R-Precision[1] to determine how successful the algorithm is performing. If within n iterations, we find all n users in the list. Then our algorithm is working with R-Precision = 100%. Therefore, any unsuccessful recommendation will decrease our R-precision. Our solution is expected to yield better results for the people from same social communities. But the test results vary correspondent to the preferences of users in which they add friends in random order and the strength between the links of friends. Our solution is expected to improve in the consequent iterations, because as we expand the graph, people tend to add their friends from the same community. This increases the number of friends who are from the same community thus increases the probability of accurate recommendations. Overall precision of FOAF-Cache is expected to be higher than 80 percent.

3. RESULTS

We tested the proposed scheme to obtain results. Our experiments were based on the 25 pre-determined users with cache sizes of 5, 7, 9, 10, 11, 13, 15, with value labels on k=5, k=10, k=15 and ranking of top 5, 7, 9, 10, 11, 13, 15 users with value labels on h=5, h=10, h=15. Some of these 25 users are used inside cache and therefore not counted in R-precision parameter. Also, results obtained are taken from not only one iteration but average of 10 iterations for the same cache size and ranking. This section has 2 subsections. First one is observations about the constant h values and different cache sizes. Second one is converse of the first one, constant cache sizes and different h values.

3.1 Experiments with Different Cache Sizes

In this section experiments are based on varying cache sizes and constant h values. Experiments in this section are made to investigate the cache size versus r-precision.

3.1.1 Experiment with h=5

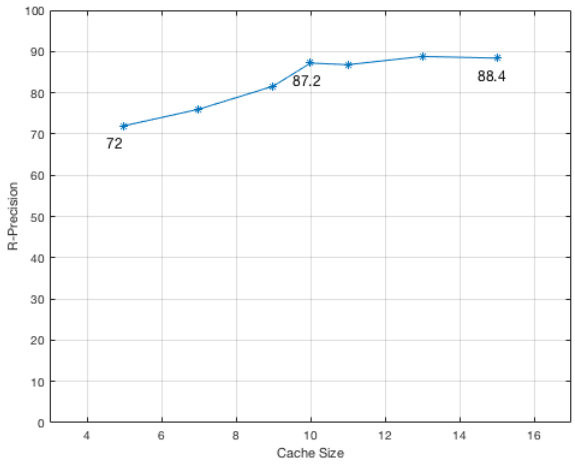

Figure 1: Test results for different cache sizes. In this test h value is constant and equal to 5. Different cache sizes and their corresponding r-precision can be seen. There are 25 relevant, target, users.

In the first experiment, h is equal to 5 which represents among top 5 recommendations of the algorithm we try to find a match for the friends we want to add. Cache sizes between 10 and 15 have similar R-Precisions of about 88 percent. However, for the cache size of 5 R-precision drops down to 72 percent. This is to be expected, since, having less cache size means having less choice and therefore chance to recommend users. Having a match in recommendation of 5 users is certainly expected to have less chance than 10 or 15 users. R-precision constantly increases as cache size grows until cache size is 10 . It is shown that cache size more than 10 gives the optimal results. Any value greater than 10 as

a cache result does not improve on accuracy but decreases the performance because cache size grows.

3.1.2 Experiment with h=10

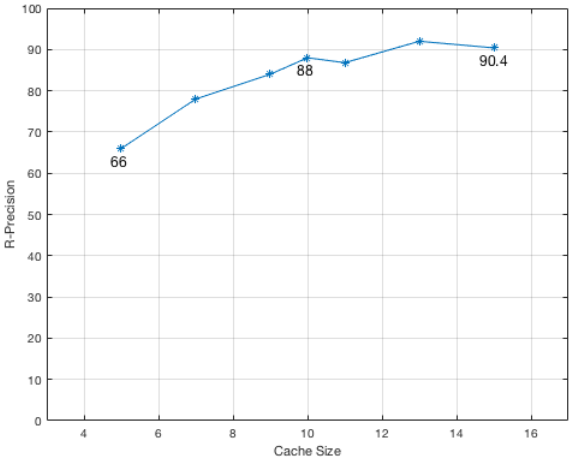

Figure 2: Test results for different cache sizes. In this test h value is constant and equal to 10. Different cache sizes and their corresponding r-precision can be seen. There are 25 relevant, target, users.

Result of the second experiment can be seen in Figure 2. h is equal to 10 which represents among top 10 recommendations of the algorithm we try to find a match for the friends we want to add. Cache sizes between 10 and 15 have similar R-precision of about 89 percent. On the other hand, for the cache size of 5 R-precision drops down to 66 percent. Having less chance in 5 users than others is expected once again. But interestingly, in this experiment, we have a decrease in performance than top 5 recommendations case for the cache size of 5. This might be explained with the different cache setups they initialized with where the elements present in the newly initialized cache hinders the performance until all of cache is filled with the users we wanted to add from the beginning. However it still was an unexpected result for us. However, between cache sizes 5 to 10 R-Precision increases dramatically and comes to rest around 9-15 band. In addition, results are similar to the previous experiment which h was equal to 5. Both experiments have the lowest results in cache size 5 and it is incrementing as cache size grows. After cache size is equal to 9-10 it is near the optimal cache size and R-Precision does not grow much after that point.

3.1.3 Experiment with h=15

In the third and final experiment can be seen in figure 3, h is equal to 15 which represents among top 15 recommendations of the algorithm we try to find a match for the friends we want to add. Cache size of between 5 and 10 yields similar results with experiment 1 and 2 slight improvement in R-precision is observed for cache size of 15. It can be interpreted as that we can find the recommendation we need among the top 5 recommendations rather than looking through top 15 recommendations. Also, the trend of similarity in R-precision for cache sizes of 10 and 15 is still present with slightly more improvement in size 15 for

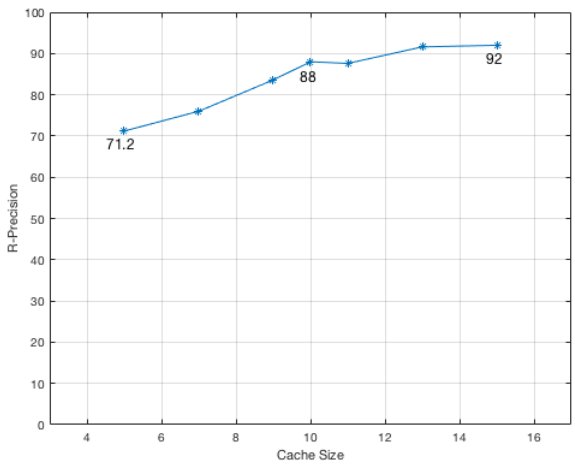

Figure 3: Test results for different cache sizes. In this test h value is constant and equal to 15. Different cache sizes and their corresponding r-precision can be seen. There are 25 relevant, target, users.

experiment 3 only.

Remarks needed to be made are about the almost similar R-precision values obtained from cache sizes of between 10 and 15. This similarity was unexpected for us. Because, we were expecting the cache size of 15 to yield significantly better results than cache size of any other number. However, improvement of only minor percentages can be explained with the curse of dimensionality where increasing the number of parameters after certain point, doesn't improve the performance of system. For our system, cache size more than 10 performs significantly well with relatively less calculation against the cache size of 15. If, the minor improvement in accuracy is considered vital for future systems, cache size can be further increased. Future work can still be conducted to determine whether the cache size for the improvement of performance is proportional to the amount of users we need to find or fixed sized.

3.2 Experiments with Different h Values

In this section experiments are based on varying number of top ranking users, h value, and constant cache sizes. Experiments in this section are made to investigate the h values versus R-Precision.

3.2.1 Experiment with cache size 5

Average R-precision obtained for different h values with cache size of 5 is around 70 percent. It proves to be not precise enough for significantly accurate friend recommendation. Although, there is a setback of 60 percent in experiment h=9. This is not surprising because as we explained in link generation, pre-determined users are assigned randomly. From this experiment we can say that overall performance for different ranking h values does not change the results much for cache size with 5. Low R-Precision is here most likely the cause of small cache size. We can say for this settings R-Precisions will vary randomly between 60-70.

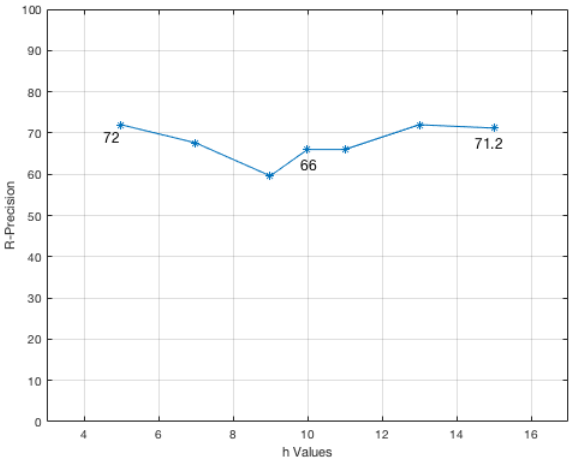

Figure 4: Test results for different h values. In this test k value is constant and equal to 5. Different h values and their corresponding r-precision can be seen. There are 25 relevant, target, users.

3.2.2 *Experiment with cache size 10*

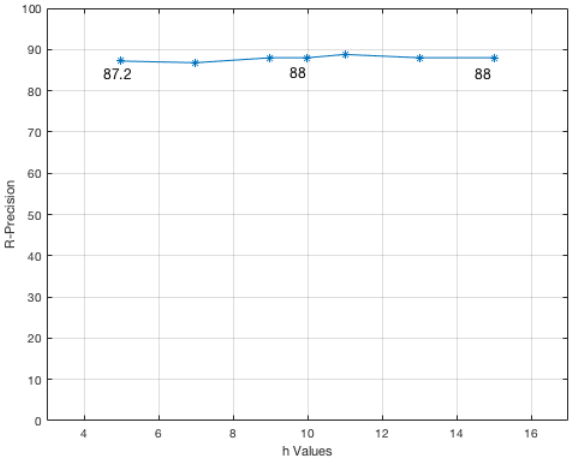

Figure 5: Test results for different h values. In this test k value is constant and equal to 10. Different h values and their corresponding r-precision can be seen. There are 25 relevant, target, users.

Average R-precision obtained for cache size of 10 is around 88 percent with almost no variance for different h values. Precision can still be improved further for friend recommendation. However, as we will see in Figure 6, curse of dimensionality starts to diminish the improvement we obtain by increasing cache size. Therefore, cache size of 10 can be regarded as the optimum solution where storage and computational efficiency are highly regarded as accuracy.

3.2.3 *Experiment with cache size 15*

Average R-precision obtained for cache size of 15 is around 90 percent with little variance for different H values. Precision increases only 2 percent against the exponentially increasing complexity of storage and computation power of P Matrix. As mentioned under Figure 5, curse of dimensionality diminishes the improvement we obtain. This cache size seems to be optimal for systems with high storage and high computational power. However, it seems minor improvements can still be obtained with increasing the cache size even further.

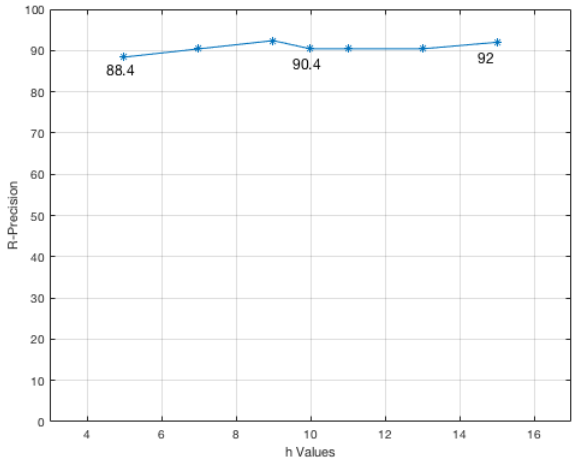

Figure 6: Test results for different h values. In this test k value is constant and equal to 15. Different h values and their corresponding r-precision can be seen. There are 25 relevant, target, users.

Overall, by evaluating experiments with h values tells us number of top ranking does not change much if the cache size is constant. In all experiments, if cache size is same, different h values have the similar values. In all 3 experiments, all h values have almost the same values except cache size is 5. In k=5 case values are randomized but again they have close R-Precisions.

4. CONCLUSION

As stated in the abstract and introduction sections of our project, this FOAF based caching approach is used for the scenario in which we assume there is a relationship between consequent users added as friend. These friends might be school friends or from the similar community. Basic assumption is that if the friends of these people have can be stored and processed together, we can recommend friends within similar communities much more easier. For that purpose, friends these recently added users have are stored in cache. Then using pairwise comparisons among the friend list of these recently added users, P-Matrix is created. P-Matrix basically stores all pairwise common friends users have. For the cache size of 5, there are 5 friends in cache and 10 pairs can be formed between these users. Common friends between these 10 pairs are stored in P-Matrix. Users who appeared most times in P-Matrix are the ones who have more friendship links with the users in cache and therefore ranked higher in recommendation list. In this ranking, top h users are recommended as friends for cache size of k and R-precision values are measured to evaluate the effectiveness of recommendation.

To make a better comparison with our results, we first marked users randomly and try to find them without making link generation. In that case R-Precision is almost 0%. Therefore, finding people from same community is hard in regular FOAF based friend recommendations. Our approach yields better results since we think that considering less people with some of them in the community will have a better chance finding other people from the same community.

The experiments are conducted to determine which parameters have more effect on the accuracy. Observation was the number of ranking top users, h value, is an inferior parameter compared to the cache size. Accuracy was mostly dominated by cache size, but if h is smaller then 5 accuracy of recommendation drops. On the other hand, another parameter k which is the size of cache changes the R-precision of recommendation significantly. For $k = 5$, R-precision is 70 percent overall, but when k is increased into 10 R-precision rates increase up to 90 percent. So storing more users in cache for comparison results with more accurate predictions about who we might want to add as friend. However, increasing cache size results with the use of exponentially increased computation power and storage. If, accuracy is considered as a priority for friend recommendation system, $k = 15$ yielded best results with 90 percent accuracy. More overall approach is to use $k = 10$, by trading off 2 percent loss in accuracy with 33 percent gain in storage and computation power.

5. FUTURE WORK

As a future study, integration of FOAF based caching with other friend recommendation algorithms can be conducted. Also, testing this algorithm with different social networks is still in the scope of discovery. Another aspect to work with caching users might be not only based on recently added friends but based on the most influential friends added recently with stronger and interconnected social networks. Since, these influential friends have stronger probability of providing the users we intend to add in the future. There are many more possible derivations for this approach. Ideas listed above are just the examples that come to mind within seconds. We hope this paper can open new doors for the improvement of friend recommendation systems.

6. REFERENCES

[1] N. Craswell. *R-Precision*, pages 2453–2453. Springer US, Boston, MA, 2009.

[2] http://konect.uni-koblenz.de/networks/facebook-wosn links. Facebook friendships, koblenz network collection. December 2017.