CS516 Project 2 CUDA Report

Corentin (Corey) Rejaud
Undergraduate Student @ Rutgers
Netid: cr482
RUID: 152008525

Abstract

Single Source Shortest Path (SSSP) algorithms are very important in practical uses like the Global Positioning System (GPS), or finding your friends and mutual friends in the Facebook graph database. In this report, we talk about parallelizing SSSP algorithms using CUDA, and specifically the Bellman-Ford SSSP algorithm along with an optimized work-efficient version of the Bellman-Ford algorithm. We parallelize this using an NVIDIA GPU using CUDA in outcore, incore, and shared memory versions. The tests will be configured by a variety of different grid and block sizes and will be tested on edges arrays sorted by either destination index or source index. These tests will be ran on 5 graphs: Pokec, WebGoogle, HiggsTwitter, Amazon0312, LiveJournal, and RoadNetCA. There are massive differences in the graphs and different configurations.

1. Introduction

Single Source Shortest Path (SSSP) algorithms arise in numerous computational disciplines, and as a result, methods for efficiently processing them are often important to the performance of many applications. Some applications include the Global Positioning System (GPS) or finding friends and mutual friends in the Facebook database. The most well known SSSP algorithm known to date is the famous Dijkstra's algorithm. It achieves a worst-case performance of O(|E| + |V| log|V|), where E is the number of edges and V is the number of vertices in the graph. This performance is very good, but it is very difficult to parallelize.

The SSSP algorithm that we will focus on in this report for the most part is the Bellman-Ford SSSP algorithm. This algorithm is slower than Dijkstra's algorithm when done sequentially, but it is very easy to parallelize. Parallelizing the Bellman-Ford algorithm is not nearly as difficult as parallelizing Dijkstra's algorithm, especially since we will only be working on graphs with positive edge weights. This will make it easier because we will not need to deal with negative weight cycles, which can be complicated to deal with, especially when trying to parallelize.

Along with parallelizing Bellman-Ford, we will also be parallelizing an optimized version of the Bellman-Ford algorithm which we call the To-Process-Edge (TPE) algorithm. In this algorithm, we will filter out of the edges that do not need to be processed in the next round of computations to minimize the time spent on computing shortest paths.

All parallelizations will be done using NVIDIA GPUs in CUDA. Tests will be done on (Block Size, Grid Size) configurations of (256, 8), (384, 5), (512, 4), (768,2), and (1024,2). Each algorithm will be tested in multiple different ways including incore, outcore, and using shared memory. Along with this, the input will be tested with a an edges array sorted by source index and an edges array sorted by destination index.

2. Parallelized Bellman-Ford Algorithm (BMF)

Bellman-Ford is an SSSP algorithm which is slower than Dijkstra's algorithm, but it is more versatile and is capable of handling graphs in which some of the edge weights are negative numbers. However, the graphs we will be testing only have positive weighted edges. The main reason that we are going to be using Bellman-Ford is because it is much easier to parallelize than Dijkstra's algorithm. Both Dijkstra's algorithm and Bellman-Ford is based on the principle of relaxation. Relaxation is when an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. As long are there are negative weight cycles, then Bellman-Ford guarantees the shortest path to all vertices from the starting

source after relaxing every edge |V| - 1 times, where |V| is the number of vertices in the graph. Since the graphs we are testing only have positive weighted edges, it is guaranteed that this algorithm will work. To speed up the process, we will be checking after each iteration if any edge has been successfully relaxed. If no edge has been relaxed during Bellman-Ford, then the process is done and we can end the iterations early.

To parallelize Bellman-Ford, we will be using CUDA kernels to relax every edge in the graph. We will call the kernel |V| - 1 times or until no edges have been successfully relaxed. In parallelizing Bellman-Ford algorithm, we exploit the parallelism of edge processing at every iteration. We can distribute edges evenly amongst different processors such that each processor in the GPU is responsible for the same number of edges.

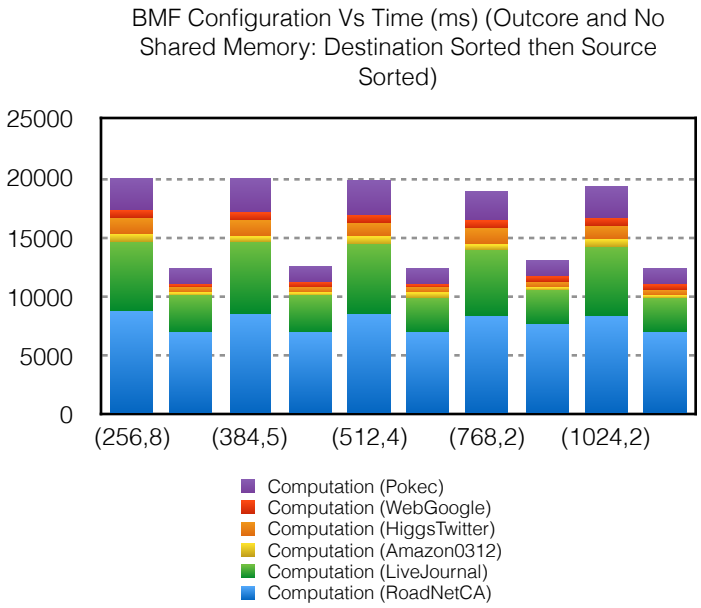
I implemented an incore version, outcore, version, and a shared memory version. The outcore version is described in sudo-code below.

```
1  kernel edge_process(L, distance_prev, distance_cur)
2  {
3    load = L.length % warp_num == 0 ? L.length/warp_num: L.length/warp_num+1;
4    beg = load*warp_id;
5    end = min(L.length, beg + load);
6    beg = beg + lane_id;
7    for ( i = beg, i < end,  i += 32 ) {
8      u = L[i].src;
9      v = L[i].dest;
10     w = L[i].weight;
11     if ( distance_prev[u] + w < distance_prev[v]
12       atomicMin(&distance_cur[v], distance_prev[u] + w);
13
14   }
15 }
16 ...
17
18 for (i from 1 to size(vertices)-1 ) {
19   edge_process(L, distance_prev, distance_cur);
20   if ( no node is changed) break;
21   else swap(distance_cur, distance_prev);
22 }
```

After each kernel call to edge_process, outside the core, it will copy distance_cur to distance_prev (it doesn't actually swap, that must be a typo). The incore version is very similar, but it uses one distance array and does that copying inside the kernel, which saves time and can skip steps. The shared memory version uses a segment-scan type method to find the minimum distances in a block before atomically updating the distance using atomicMin.

1. Results (Outcore, no Shared memory)

The results to algorithm we very positive. It seems to be the second quickest parallelization of normal Bellman-Ford algorithm.



When the edges array is sorted by destination, it seems to be slow relative to when the edges array is sorted by source. This seems to be because the atomicMin command isn't locking up certain threads because the destination indices are much more spread out within a block. When the edges array is sorted by destination, then the atomicMin command will lock up threads within the same block since blocks run in a queue and the destination indices are bunched up together.

As far as the grid and block size configurations go, there doesn't seem to be a huge difference in what the configuration is. This is mainly because this approach does not use shared memory or any segment scan approaches, which benefit from these different configurations. It seems like (1024,2) is the fastest configuration when the edges array is sorted by source index, but it is not extremely noticable. This is mostly because it has the most amount of threads running at once, and there is less need to switch blocks, since there are only two blocks.

2. Pros (Outcore, no Shared memory)

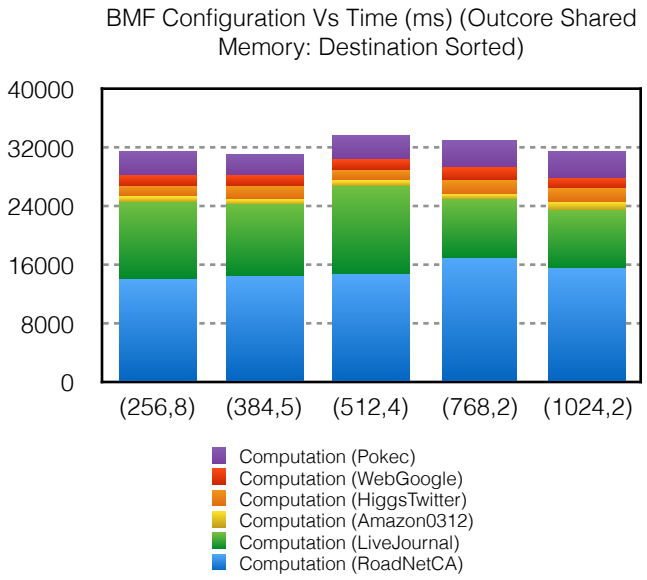
Some pros about the outcome with no shared memory version is that it is extremely easy to implement. By far the easiest out of all the other versions. The speed of this is very reliable, especially if you sort the edges array by source index.

3. Cons (Outcore, no Shared memory)

Some cons about the outcome with no shared memory version is that it is not consistent. If the edges array is sorted by destination, then the time to compute this algorithm can take up to 2x longer than if the edges array was sorted by source. Other than that, there is nothing too bad about this approach. It seems to be the second quickest approach for normal Bellman-Ford.

4. Results (Outcore with Shared memory)

The results to this version of BMF was not very positive. It seems to be the slowest parallelization of normal Bellman-Ford algorithm by far.



This algorithm only seems to work well with the Pokec graph compared to the other outcore version. This may be because there are a lot of strongly connected components, which is a benefit to shared memory approaches. This approach was only used on the edges array being sorted by destination, because if it were to be sorted by source, then this approach would take way too long.

5. Pros (Outcore with Shared memory)

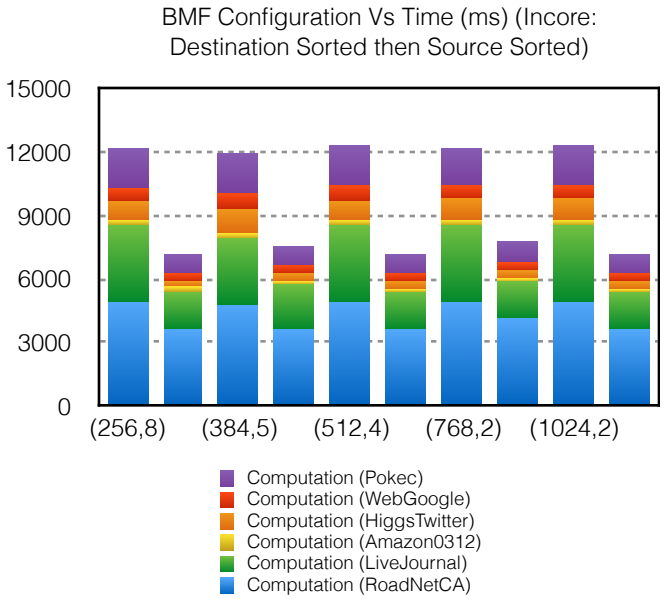
This version of BMF worked well on graphs with a high frequency of strongly connected components and widely connected components, like the social network, Pokec. However, that is all to say that is good about this implementation.

6. Cons (Outcore with Shared memory)

Firstly, this implementation was a lot more difficult to implement than the other outcore version and the incore version because there were a lot more moving parts. Coding this solution took a whole day to get working, and yet it is still very slow. Secondly, Working on larger graphs like RoadNetCA and LiveJournal, this approach did terrible. It took roughly 16000 ms to compute, which is up to 5x longer than it's other implementations, normal outcore and incore.

7. Results (Incore)

The results for this version were extremely positive. It seems to be the fastest in the normal Bellman-Ford algorithm section.



As you can see, the overall times are the fastest by far. When the edges array is sorted by destination, it ties up with the outcore version when the edges array is sorted by source. But when the edges array is sorted by source, it is 2x faster than if it were sorted by destination. The speeds on this algorithm are tremendous. Again there does not seem to be much difference on the configurations of grid and block sizes, but again, like the normal outcore version, it seems to be slightly quicker when using (1024,2).

8. Pros (Incore)

This algorithm version of Bellman-Ford is slightly more difficult to implement than the nor-

mal outcore version, but the performance increase is very much worth it. This version takes on a 2x increase in speed over the normal outcore version and a 3x speedup over the outcore shared memory version.

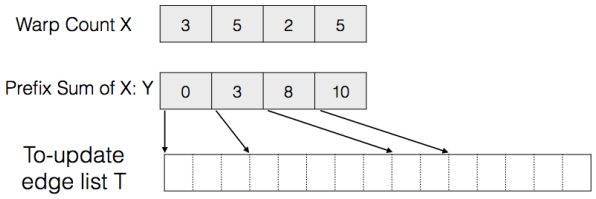
9. Cons (Incore)

The only con about this approach is that is inconsistent with how the edges array is sorted. If the edges array is sorted by destination, then it is 2x slower than if it were sorted by source. But even at it's slowest, it still performs quicker than the normal outcore version and the outcore shared memory version.

3. Parallelized To-Process-Edge Algorithm (TPE)

The To-Process-Edge (TPE) algorithm is an optimization on the normal Bellman-Ford algorithm. At each iteration, Bellman-Ford relaxes every edge in the graph regardless if it even needs to be relaxed. The TPE algorithm filters out the edges that do not need to be relaxed after each iteration and the computation stage only relaxes the edges that need to processed. This will be done in CUDA as well with the same grid and block size configurations, but we won't be using shared memory, just an incore and outcore version.

To figure out which edges need to be filtered out, we try to find the edges whose starting point did not change in the last iteration and only keep the edges that might lead to a node distance update. We refer to the edges that need to be checked as to-process edges. To achieve this, I used a parallel prefix sum operation.



Before any computations are done, we first need to find which edges need to be processed per warp in a to-process edges array. I do this by calling a new kernel which simply sifts through each edge and checks to see if distance at the source changes (we cannot do this in the computation kernel since we only change the distances at the destination index, so not all distances at the source indices may have changed yet).

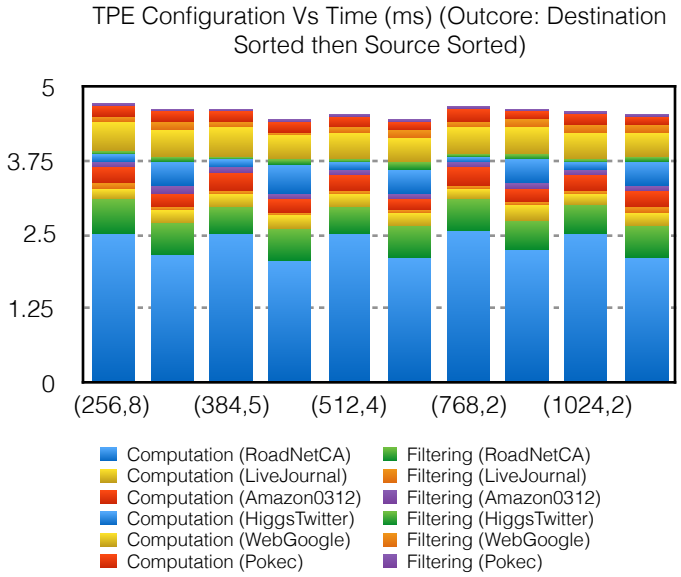
After we have this array of to-process edges by warp id, we move onto the filtering step, which is another kernel function. We limit this kernel function to a grid size of 1, and a block size of 64, since we only ever have a maximum of 64 warps based on our configurations. We choose 64 every time because it makes it much easier to perform a parallel prefix sum operation, then if the number of warps aren't actually 64 (like (768,2) which has 48 warps), then we simply remove the padded warps at the end after the parallel prefix sum operation is done.

Inside this filtering kernel function, we perform a block level parallel prefix sum to get the offset of every warp's first to-process edge in the final to-process edge list which we denote as T. The size of the array T is the total number of edges that need to be processed. We then let every warp copy its identified to-process edge index to the array T. if a warp's offset is y[i], and it has detected k to-process edges (not necessarily contiguous), then it will copy these k edges to T[y[i]]:T[y[i] + k - 1].

Once we have this array T, we call the computation kernel and balance the work load based on the warp id. There will generally be way less work to do in this computation kernel since it won't need to check every edge in the graph, only the edges in T based on the warp id.

1. Results (Outcore)

The results to this version of TPE was very fast! The computation speeds are insanely



quicker than any of the normal BMF versions above.

The speed of filtering seems to be anywhere from 0.04 to 1 ms and the speed of computing seems to be anywhere from 0.2 to 2ms, which is vastly quicker than normal BMF. It does not seem like there is a huge difference based off the configuration of grid and block sizes, but it seems like it the quickest

Overall, it does not seem like there is a huge difference between the computation speeds of whether the edges array is sorted by destination or source. But it seems like the computation speed is quicker if the edges array is sorted by source for the RoadNetCA graph. This may be because of how the edges are partitioned for each warp in the to-update edge list T, but it not noticeable. However, it seems like the computation speed is slower if the edges array is sorted by source for the HiggsTwitter graph. Again, this must be how the edges are partitioned in the T array, but there doesn't seem to be a huge or noticeable difference.

2. Pros (Outcore)

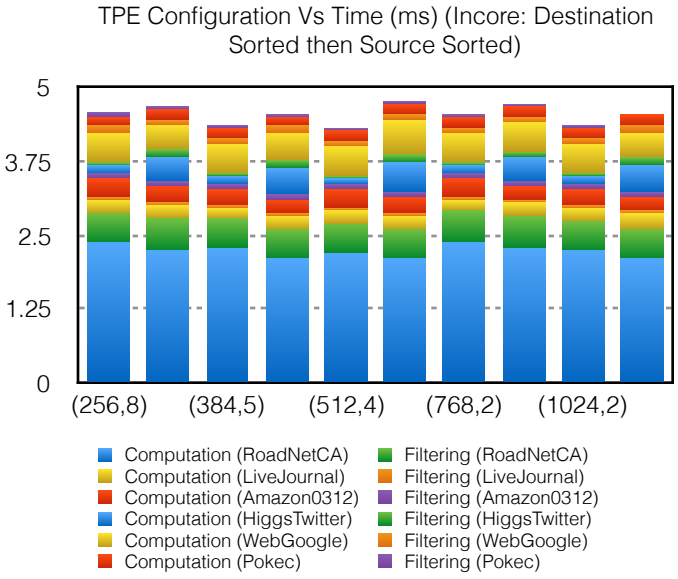
This algorithm version is way faster than the normal BMF versions above. It blows the computation speeds out of the water. It seems to be at least 1000x faster than the normal BMF versions if we compare only the computation speeds.

3. Cons (Outcore)

This algorithm version is so much more difficult to implement than any of the BMF versions above. There are so many moving parts and many lines of code to write, but given the performance, I would say it is worth it.

4. Results (Incore)

The results to this version of TPE was also very fast! The computation speeds are slightly faster than the TPE outcore version, but not a huge amount of difference. It seems to have the same differentiation in speeds based off the whether the edges array is sorted by destination or source as the TPE outcore version, so there is not much more that is different about this approach. However, it seems to be just slightly quicker.



5. Pros (Incore)

This is the fastest version of TPE, but only marginally. It is also at least 100x faster than the normal BMF versions if we only compare the computation speeds.

6. Cons (Incore)

This algorithm version is just as difficult as the outcore TPE version for the most part. There are a lot of moving parts but given the performance it is definitely worth it. It is a big more inconsistent than the outcore version based on the graphs we tested, but it seems to be bit coincidence because it can go both ways.

4. Conclusion

It seems like the quickest parallelized computation time is if we implement the TPE outcore or incore versions. There isn't any noticeable difference between those two but there is significant difference between the TPE versions and the normal BMF versions. If you would want the quickest computation performance, then I would definitely recommend implementing a TPE version over any normal BMF version. The pros of TPE vastly over-weigh the cons of TPE, so it is always in your benefit to implement a TPE version than implement a BMF version, when parallelizing. The BMF outcore shared memory version seems to have the worst performance. In order of perfor-

mance from worst performance to best performance, I would list it as: BMF outcore shared memory, BMF outcore no shared memory, BMF incore, TPE outcore, then TPE incore. The below are all of my data points if you would like to look over them.

				(256,8)	(256,8)	(384,5)	(384,5)	(512,4)	(512,4)	(768,2)	(768,2)	(1024,2)	(1024,2)
Graph Name	**Algo**	**Core (In/Out)**	**Smem (Yes/No)**	**Time dest sorted**	**Time src sorted**	**Time dest sorted**	**Time src sorted**	**Time dest sorted**	**Time src sorted**	**Time dest sorted**	**Time src sorted**	**Time dest sorted**	**Time src sorted**
RoadNetCA	BMF	Out	No	8633	6815	8511	6924	8341	6819	8172	7482	8228	6829
RoadNetCA	BMF	Out	Yes	14173	X	14629	X	14754	X	16903	X	15709	X
RoadNetCA	BMF	In	X	4810	3566	4743	3654	4808	3556	4897	4051	4866	3562
RoadNetCA	TPE	Out	X	2.494 | 0.579	2.133 | 0.539	2.496 | 0.506	2.052 | 0.521	2.487 | 0.508	2.098 | 0.518	2.565 | 0.529	2.213 | 0.531	2.5 | 0.54	2.101 | 0.536
RoadNetCA	TPE	In	X	2.369 | 0.517	2.225 | 0.541	2.282 | 0.477	2.076 | 0.501	2.209 | 0.486	2.109 | 0.508	2.392 | 0.509	2.294 | 0.518	2.237 | 0.497	2.15 | 0.515
LiveJournal	BMF	Out	No	5998	3108	5969	3127	5978	3090	5681	3047	5878	3029
LiveJournal	BMF	Out	Yes	10500	X	9730	X	11977	X	8072	X	7962	X
LiveJournal	BMF	In	X	3678	1860	3206	2132	3696	1862	3591	1818	3718	1853
LiveJournal	TPE	Out	X	0.214 | 0.057	0.238 | 0.057	0.208 | 0.044	0.232 | 0.044	0.208 | 0.046	0.235 | 0.048	0.208 | 0.043	0.247 | 0.047	0.205 | 0.046	0.25 | 0.048
LiveJournal	TPE	In	X	0.218 | 0.057	0.231 | 0.06	0.214 | 0.045	0.227 | 0.047	0.22 | 0.041	0.224 | 0.048	0.213 | 0.048	0.224 | 0.046	0.216 | 0.047	0.249 | 0.046
Amazon0312	BMF	Out	No	620	303	613	303	608	299	583	298	592	291
Amazon0312	BMF	Out	Yes	802	X	792	X	833	X	900	X	913	X
Amazon0312	BMF	In	X	234	140	256	132	254	140	235	138	253	139
Amazon0312	TPE	Out	X	0.274 | 0.099	0.235 | 0.095	0.342 | 0.09	0.236 | 0.086	0.291 | 0.094	0.205 | 0.083	0.29 | 0.093	0.233 | 0.089	0.27 | 0.083	0.276 | 0.087
Amazon0312	TPE	In	X	0.276 | 0.103	0.266 | 0.099	0.257 | 0.088	0.24 | 0.083	0.306 | 0.091	0.252 | 0.096	0.289 | 0.098	0..247 | 0.088	0.299 | 0.096	0.246 | 0.093
HiggsTwitter	BMF	Out	No	1300	461	1290	461	1292	453	1207	450	1247	440
HiggsTwitter	BMF	Out	Yes	1512	X	1533	X	1565	X	1732	X	1718	X
HiggsTwitter	BMF	In	X	967	341	1083	342	969	342	1054	382	966	336
HiggsTwitter	TPE	Out	X	0.128 | 0.053	0.428 | 0.115	0.135 | 0.038	0.493 | 0.11	0.12 | 0.037	0.424 | 0.104	0.129 | 0.04	0.418 | 0.095	0.12 | 0.037	0.416 | 0.108
HiggsTwitter	TPE	In	X	0.139 | 0.048	0.399 | 0.122	0.148 | 0.034	0.476 | 0.123	0.122 | 0.034	0.495 | 0.116	0.136 | 0.057	0.396 | 0.103	0.124 | 0.037	0.444 | 0.11
WebGoogle	BMF	Out	No	635	343	715	319	679	362	646	342	633	341
WebGoogle	BMF	Out	Yes	1433	X	1455	X	1480	X	1668	X	1657	X
WebGoogle	BMF	In	X	635	343	758	319	679	361	645	342	633	340
WebGoogle	TPE	Out	X	0.491 | 0.118	0.442 | 0.118	0.482 | 0.11	0.389 | 0.098	0.453 | 0.106	0.442 | 0.11	0.451 | 0.101	0.463 | 0.112	0.486 | 0.109	0.418 | 0.11
WebGoogle	TPE	In	X	0.504 | 0.115	0.414 | 0.115	0.49 | 0.111	0.472 | 0.101	0.472 | 0.105	0.606 | 0.11	0.471 | 0.107	0.478 | 0.115	0.496 | 0.107	0.446 | 0.113
Pokec	BMF	Out	No	2856	1320	2830	1327	2847	1315	2688	1304	2818	1296
Pokec	BMF	Out	Yes	3101	X	3169	X	3208	X	3632	X	3554	X
Pokec	BMF	In	X	1874	900	1867	902	1875	898	1804	1022	1870	897
Pokec	TPE	Out	X	0.182 | 0.057	0.174 | 0.058	0.177 | 0.043	0.16 | 0.04	0.185 | 0.047	0.158 | 0.045	0.182 | 0.046	0.162 | 0.046	0.173 | 0.044	0.163 | 0.041
Pokec	TPE	In	X	0.177 | 0.054	0.175 | 0.055	0.183 | 0.044	0.164 | 0.043	0.179 | 0.045	0.167 | 0.044	0.177 | 0.044	0.174 | 0.044	0.178 | 0.043	0.162 | 0.043