HyPR: Hybrid Page Ranking on Evolving Graphs

Hemant Kr Giri
Roll No:1802010
M.Tech-CSE

Under the Guidance of
Dr. Dip Sankar Banerjee

Introduction to Page Rank

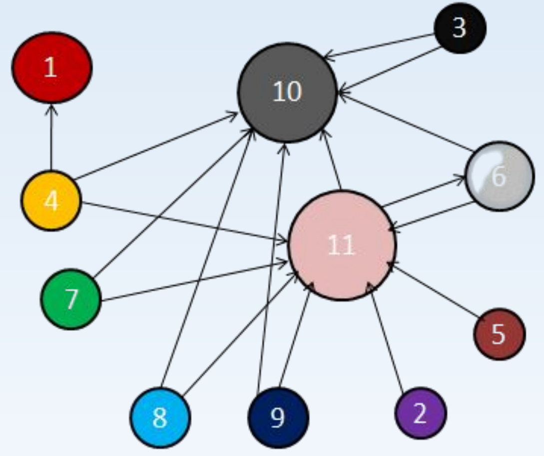
In short PageRank is a “vote”, by all the other pages on the web, about how important a page is.[2]

PageRank (PR) is an algorithm used by google search to rank web pages in their search engine results.[2]

PR is named after Larry Page, one of the founders of google.[2]

PR is a way of measuring the importance of website pages.[2]

Page rank is important because it’s one of the factors a search engine like google takes into account when it decides which results to show at the top of its search engine listings.



A Simple Version of PageRank (Power Iteration method)

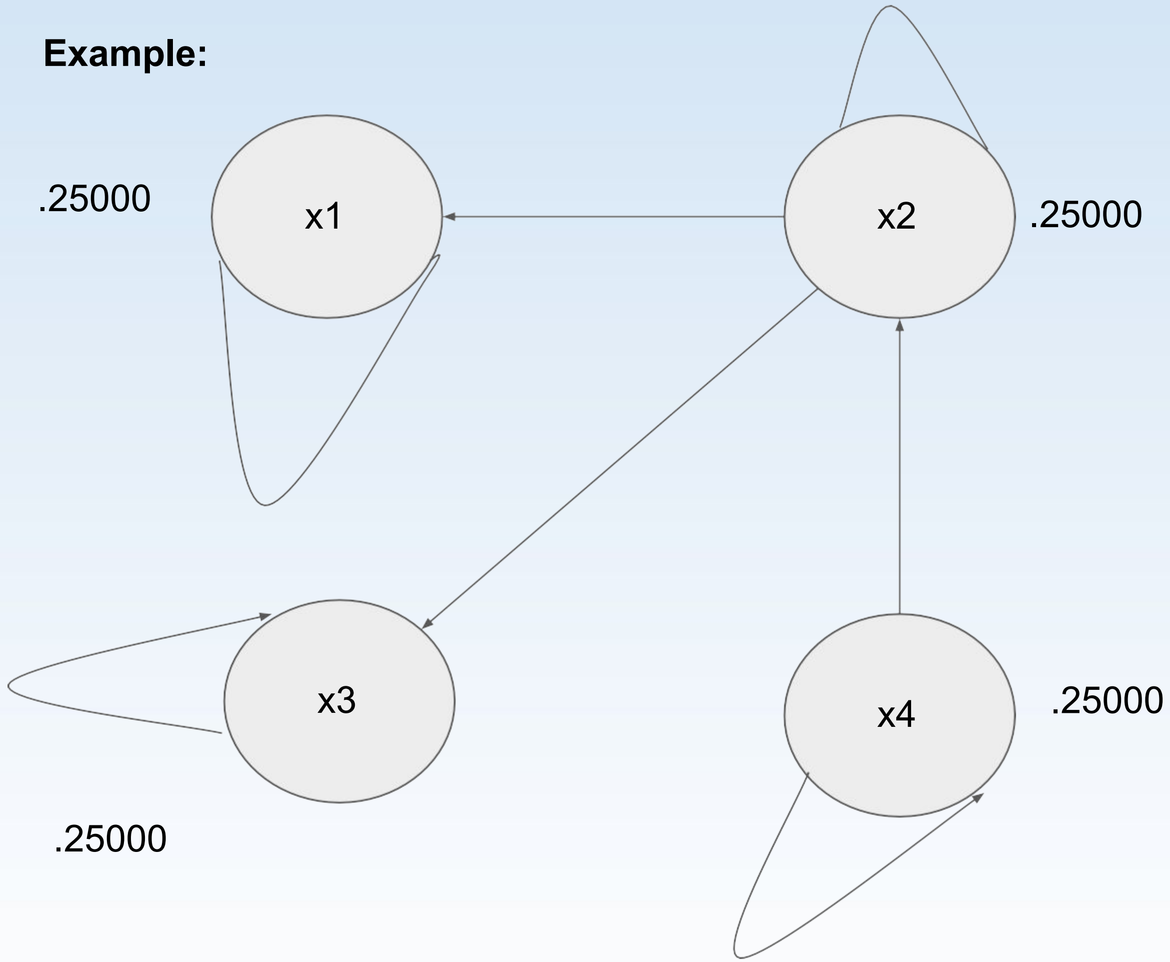
1. Suppose there are N web pages
2. Initialize: $r^{(0)} = [1/N,....,1/N]^T$
3. Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
4. Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
5. Stop when $|r^{(t+1)} – r^{(t)}|_1 < \varepsilon$

$|x|_1 = \sum_{1<=i<=N} |x_i|$ is the L_1 norm
Can use any other vector norm, e.g., Euclidean

$$r_j = \sum_{i \to j} \frac{r_i}{\mathrm{d_i}}$$

d(i).... out-degree of node i.
r(j)..... Rank of jth node.

Example:



Adjacency Matrix

	x1	**x2**	**x3**	**x4**
x1	1	0	0	0
x2	1	1	1	0
x3	0	0	1	0
x4	0	1	0	1

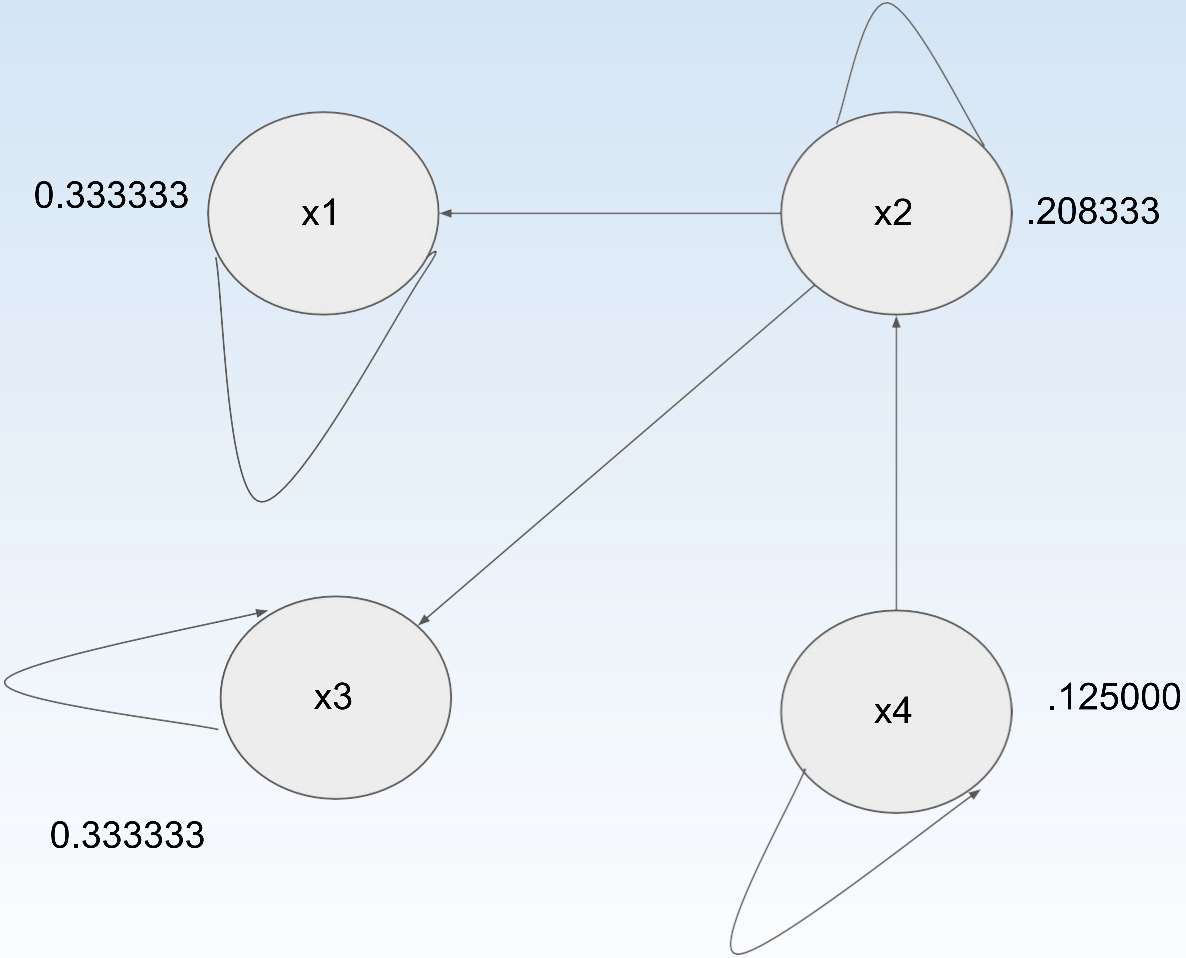
Computation of Pages having at least one outlink

PageRank Computation(after 6 iteration)

Adjacency Matrix

1	0	0	0
1	1	1	0
0	0	1	0
0	1	0	1

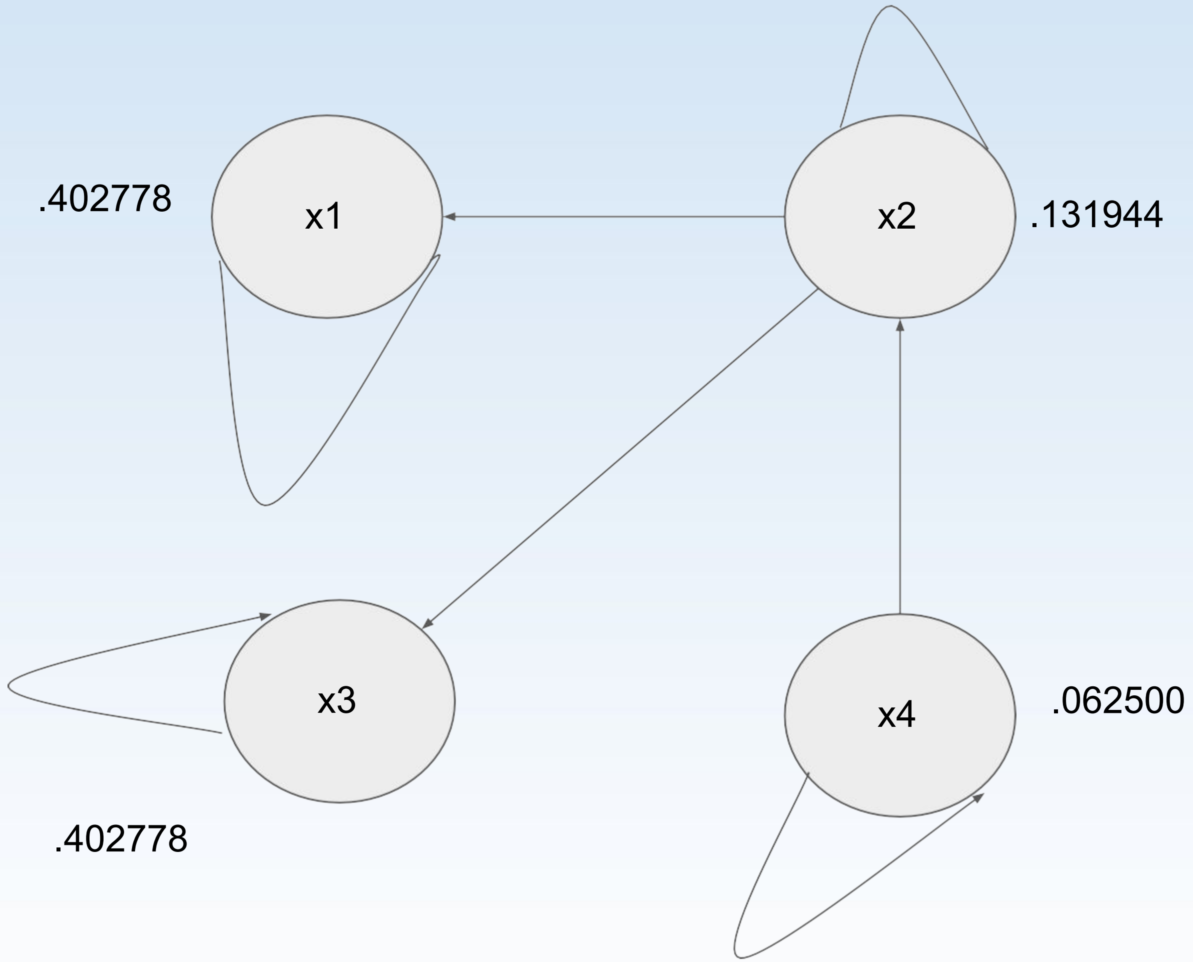
Iterations	x1	x2	x3	x4
0	0.250000	0.250000	0.250000	0.250000
1	0.333333	0.208333	0.333333	0.125000
2	0.402778	0.131944	0.402778	0.062500
3	0.446759	0.075231	0.446759	0.031250
4	0.471836	0.040702	0.471836	0.015625
5	0.485404	0.021380	0.485404	0.007812
6	0.492530	0.011033	0.492530	0.003906



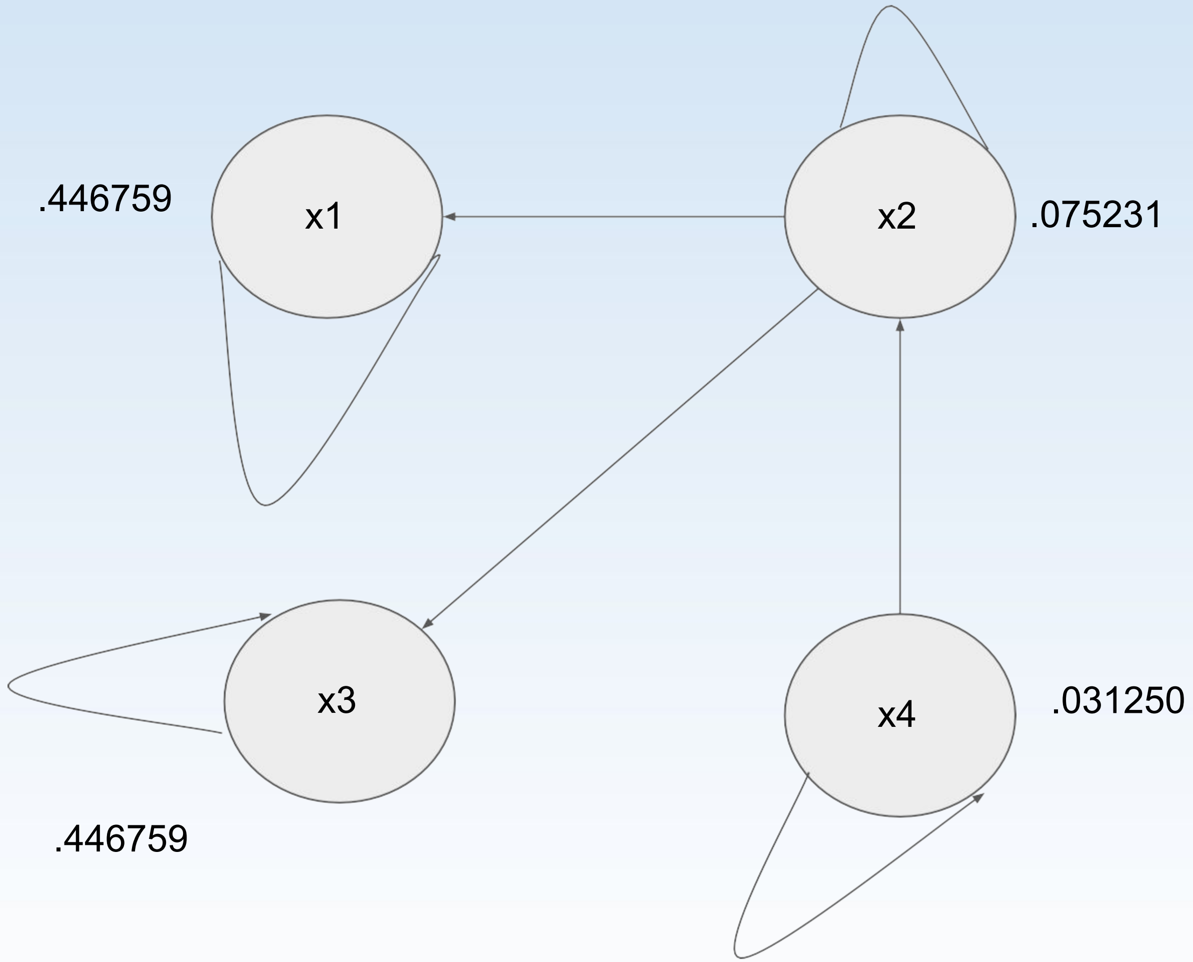
Convergence criteria

$|r^{(t+1)} - r^{(t)}|_1 < 10^{-5}$

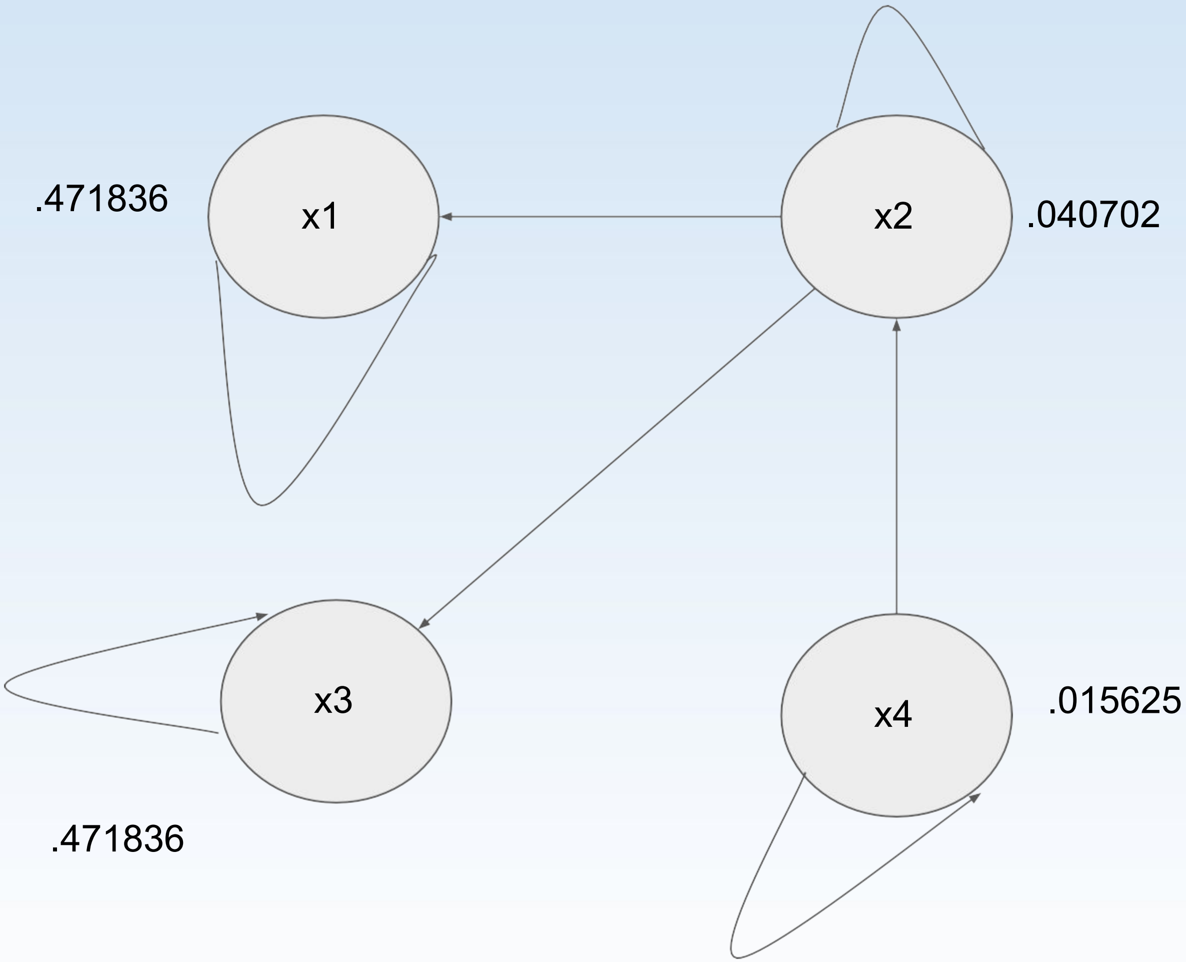
x1	x1/1 + x2/3 = .333333
x2	x2/3 + x4/2 = .208333
x3	x2/3 + x2/1 = .333333
x4	x4/2 = .125000



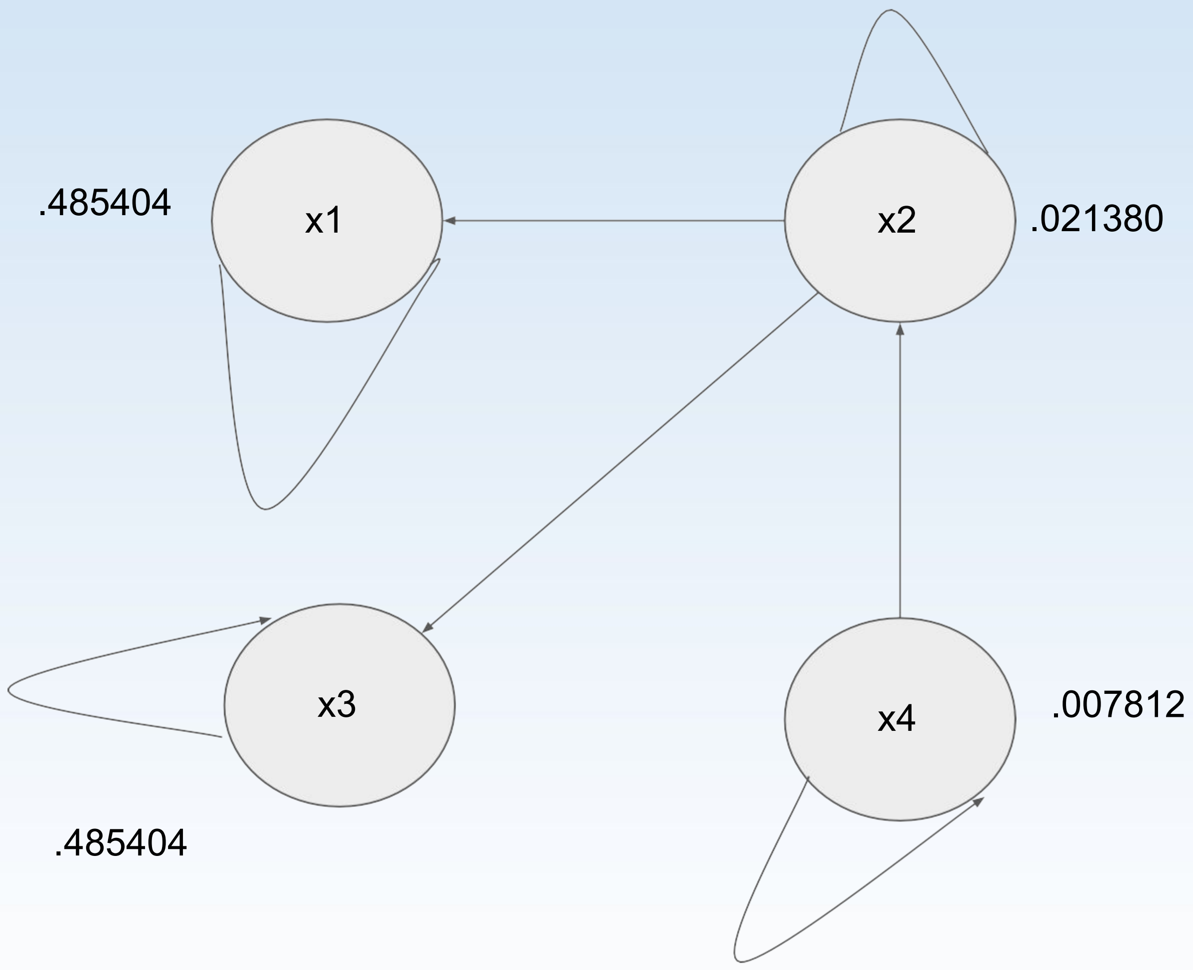
x1	x1/1 + x2/3 = .402778
x2	x2/3 + x4/2 = .131944
x3	x2/3 + x2/1 = .402778
x4	x4/2 = .062500



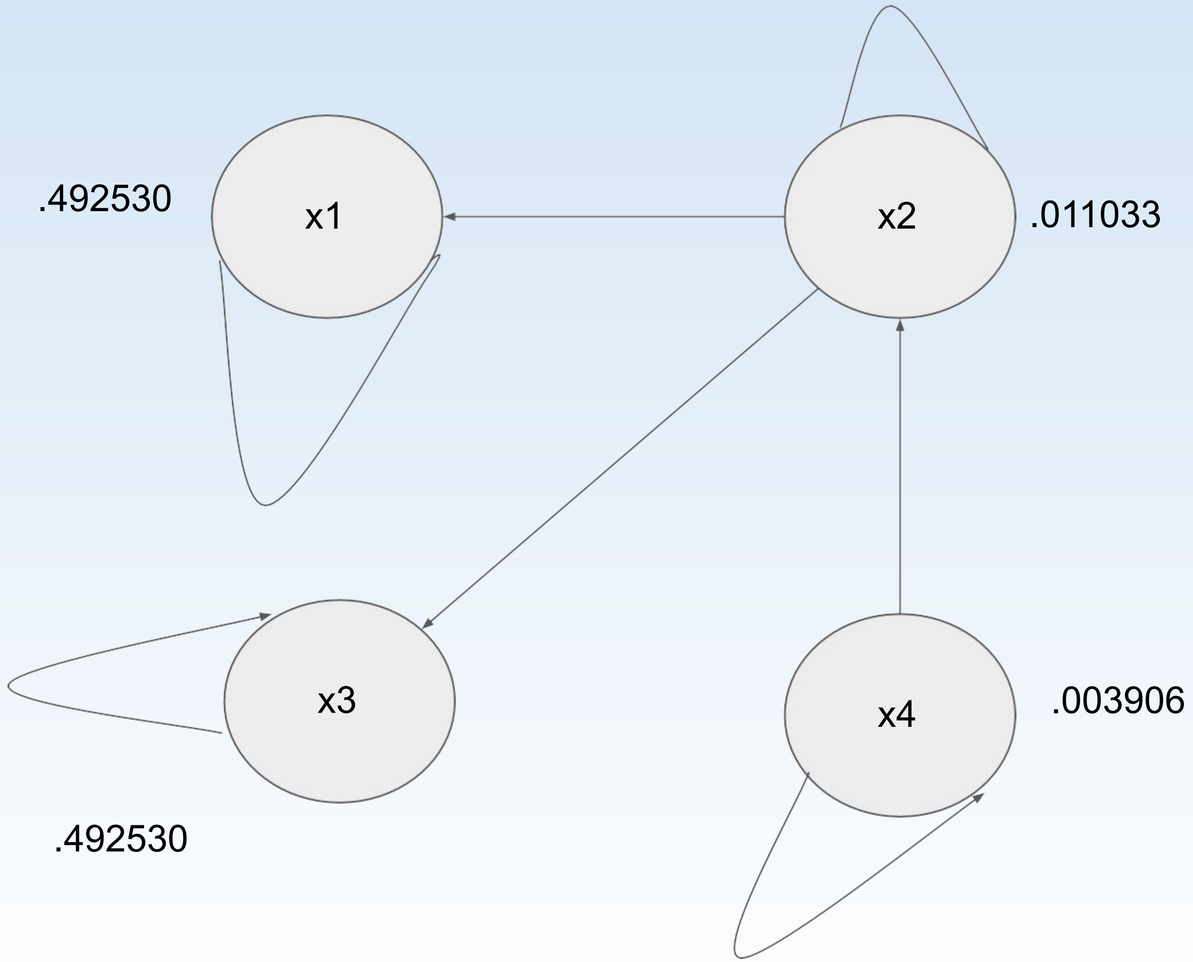
x1	x1/1 + x2/3 = .446759
x2	x2/3 + x4/2 = .075231
x3	x2/3 + x2/1 = .446759
x4	x4/2 = .031250



x1	x1/1 + x2/3 = .471836
x2	x2/3 + x4/2 = .040702
x3	x2/3 + x2/1 = .471836
x4	x4/2 = .015625



x1	x1/1 + x2/3 = .485404
x2	x2/3 + x4/2 = .021380
x3	x2/3 + x2/1 = .485404
x4	x4/2 = .007812



x1	x1/1 + x2/3 = .492530
x2	x2/3 + x4/2 = .011033
x3	x2/3 + x2/1 = .492530
x4	x4/2 = .003906

Problem With the Original Formula of PageRank

There were two problem:
1) **Dead End** It drains the pagerank to zero after some iteration.

2) **Spider Trap** It does not converge.

To overcome this challenge, Brin and Page introduce some adjustments:-

- Firstly, they replaced nodes with out-degree zero (called dangling nodes) by nodes linking to all other nodes and ,

- Secondly, they added a damping factor which influencing the random walks of the random surfer process of the graph.

Computation Problem with pages without any outLink

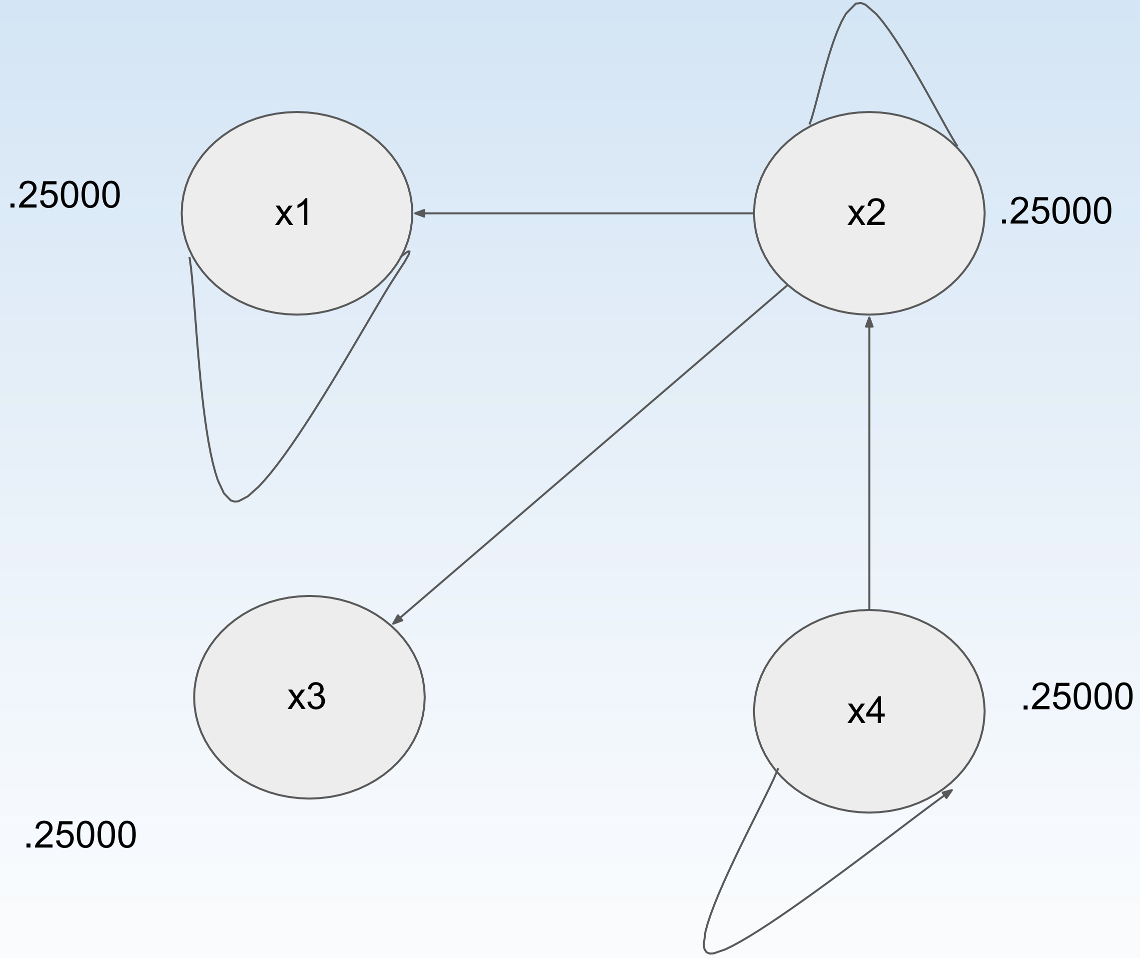
Adjacency Matrix

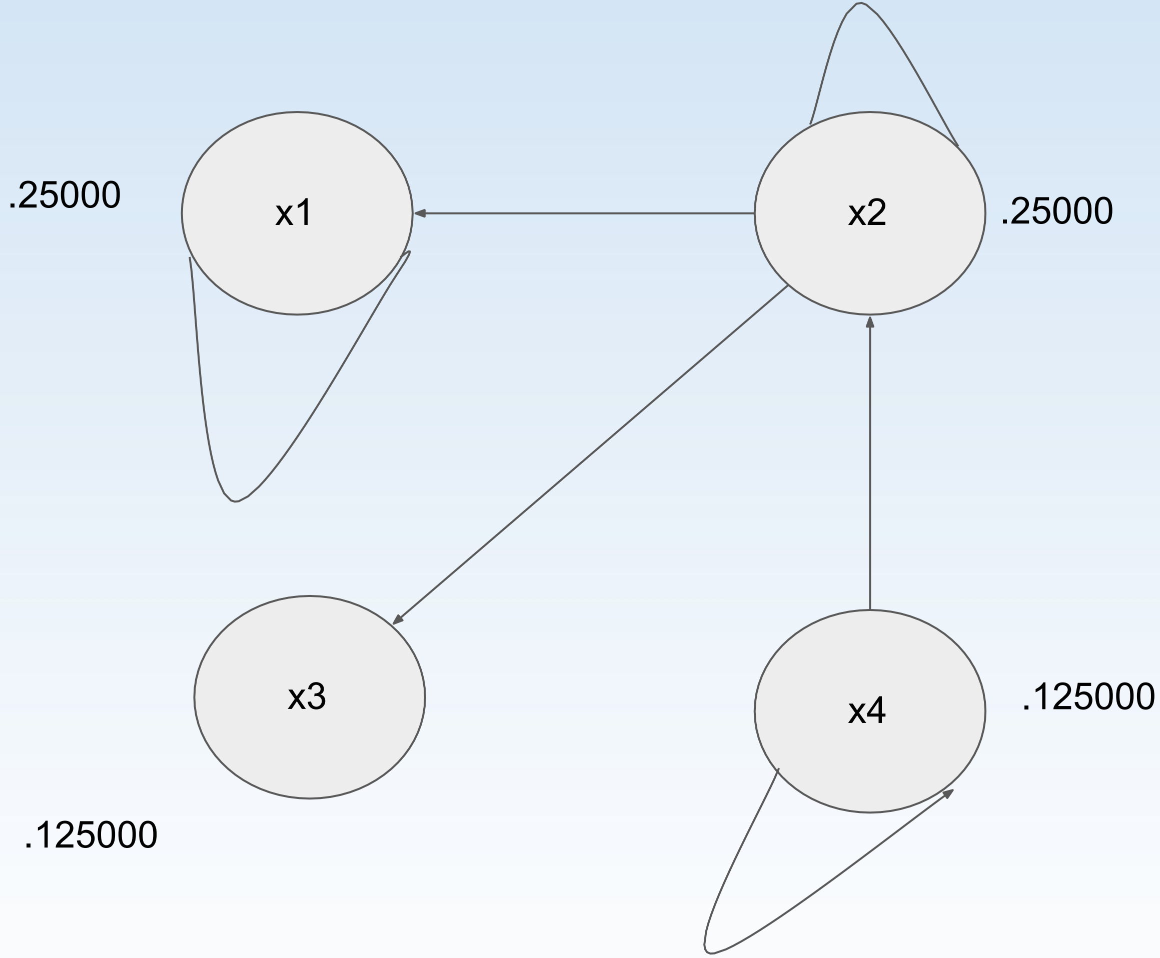
1	0	1	0
1	1	1	0
0	0	0	0
0	1	0	1

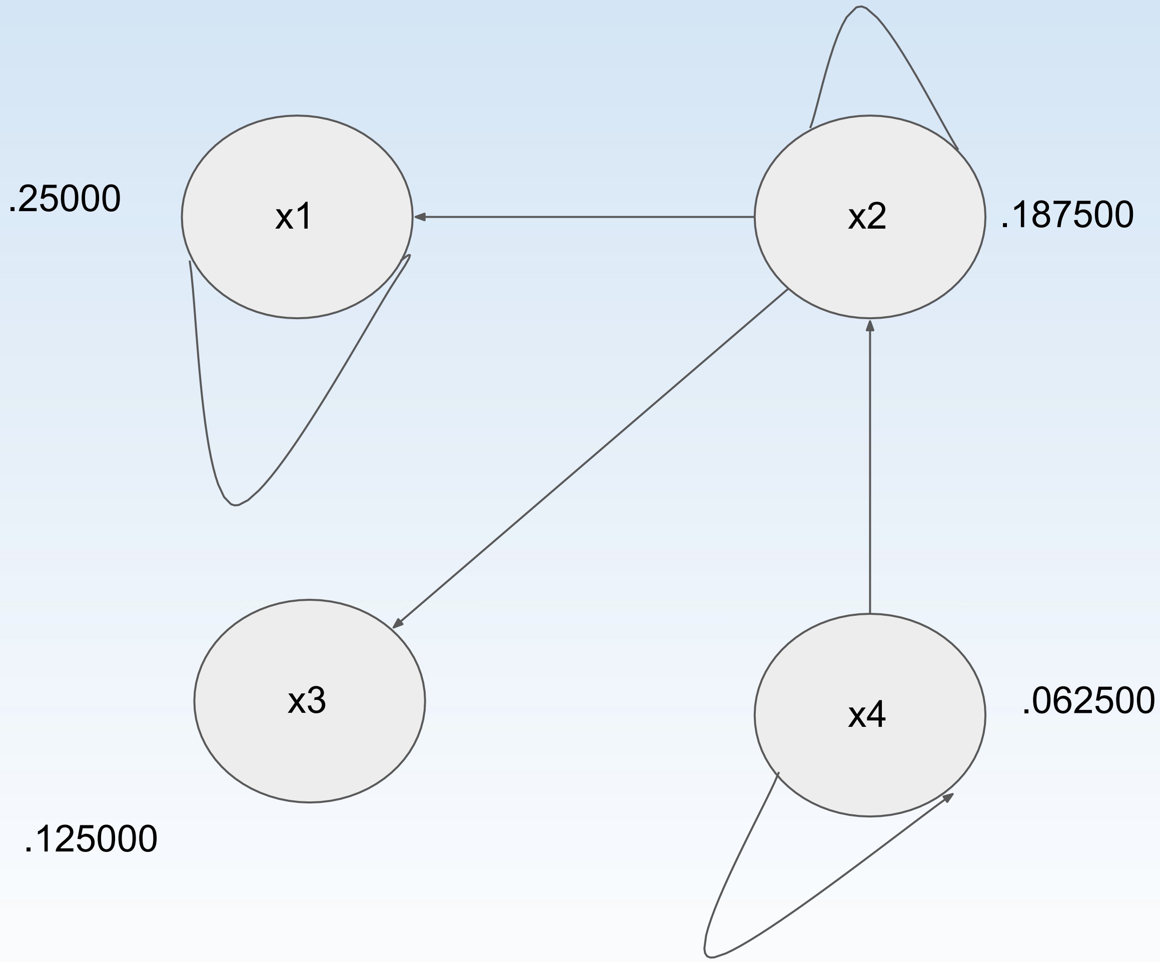
PageRank Computation(after 21 iteration)

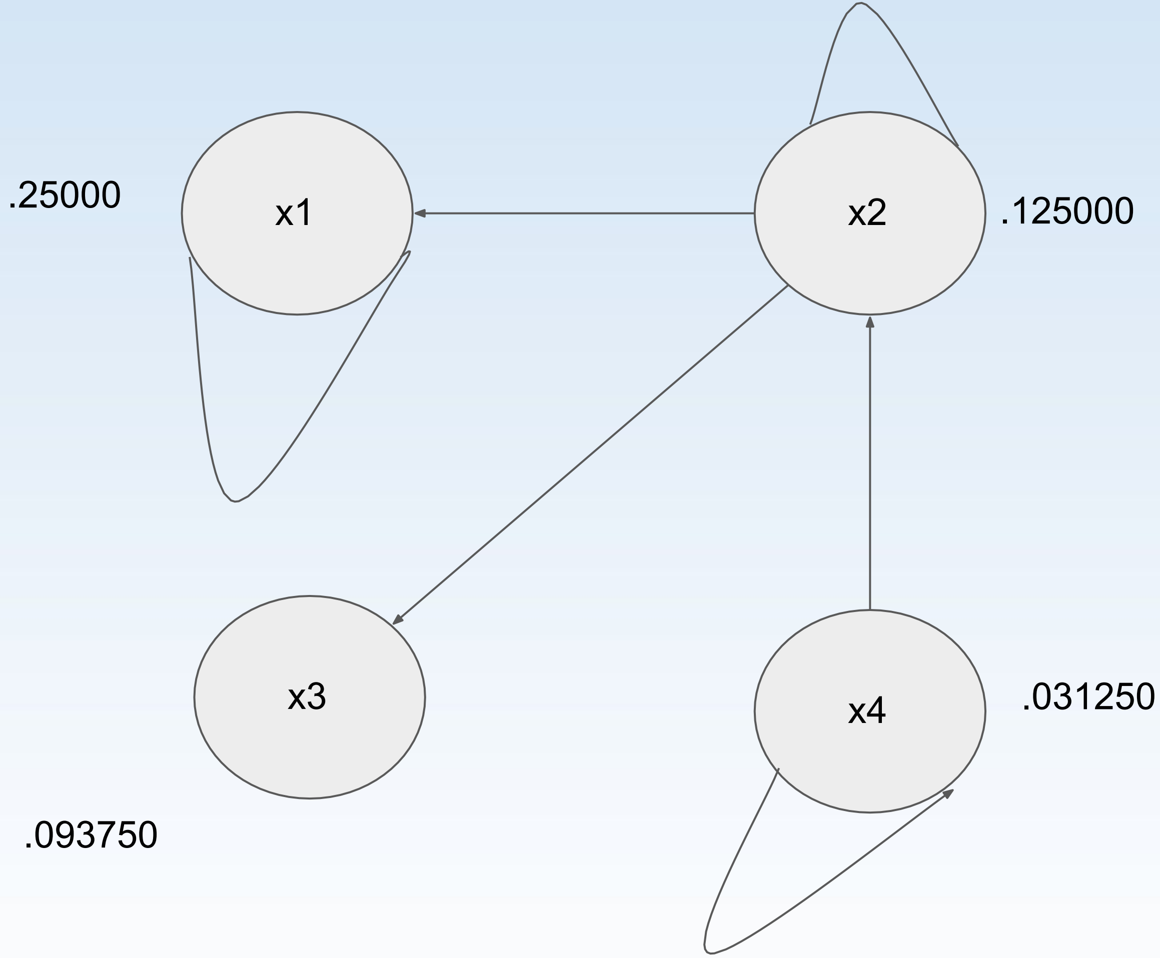
Iterations	x1	x2	x3	x4
0	.2500000	.2500000	.2500000	.2500000
1	0.333333	0..208333	0.083333	0.125000
2	0.402778	0.131944	0.069444	0.062500
3	0.446759	0.075231	0.043981	0.031250
4	0.471836	0.040702	0.025077	0.015625
……...	0.000000	0.000001	0.000001	0.000000
21	0.000000	0.000000	0.000000	0.000000

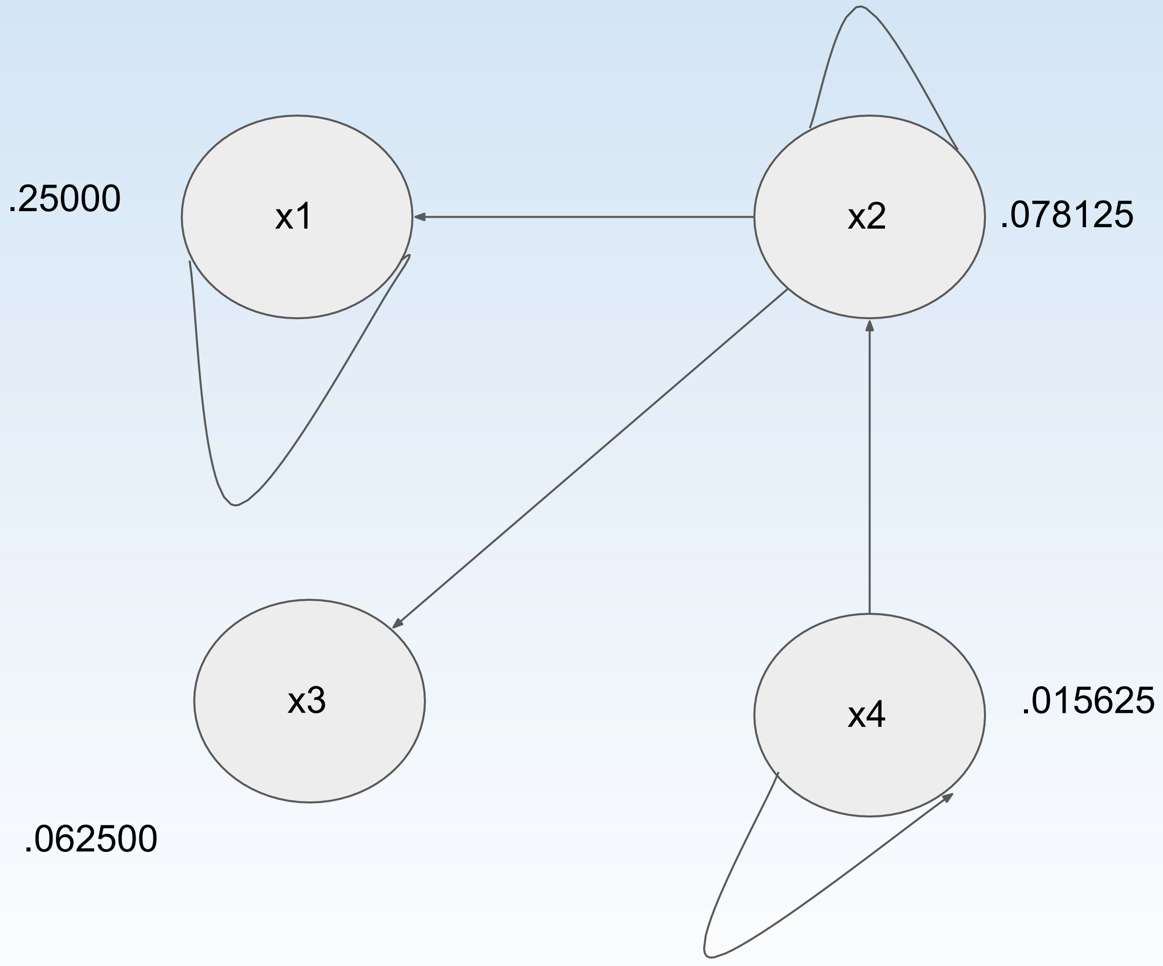
There are many pages without any out-links in the graph. This results in that there will be PageRank value sinking to zero at the end of iterative process hence, it becomes hard to rank web pages using PageRank when these values are mostly 0.

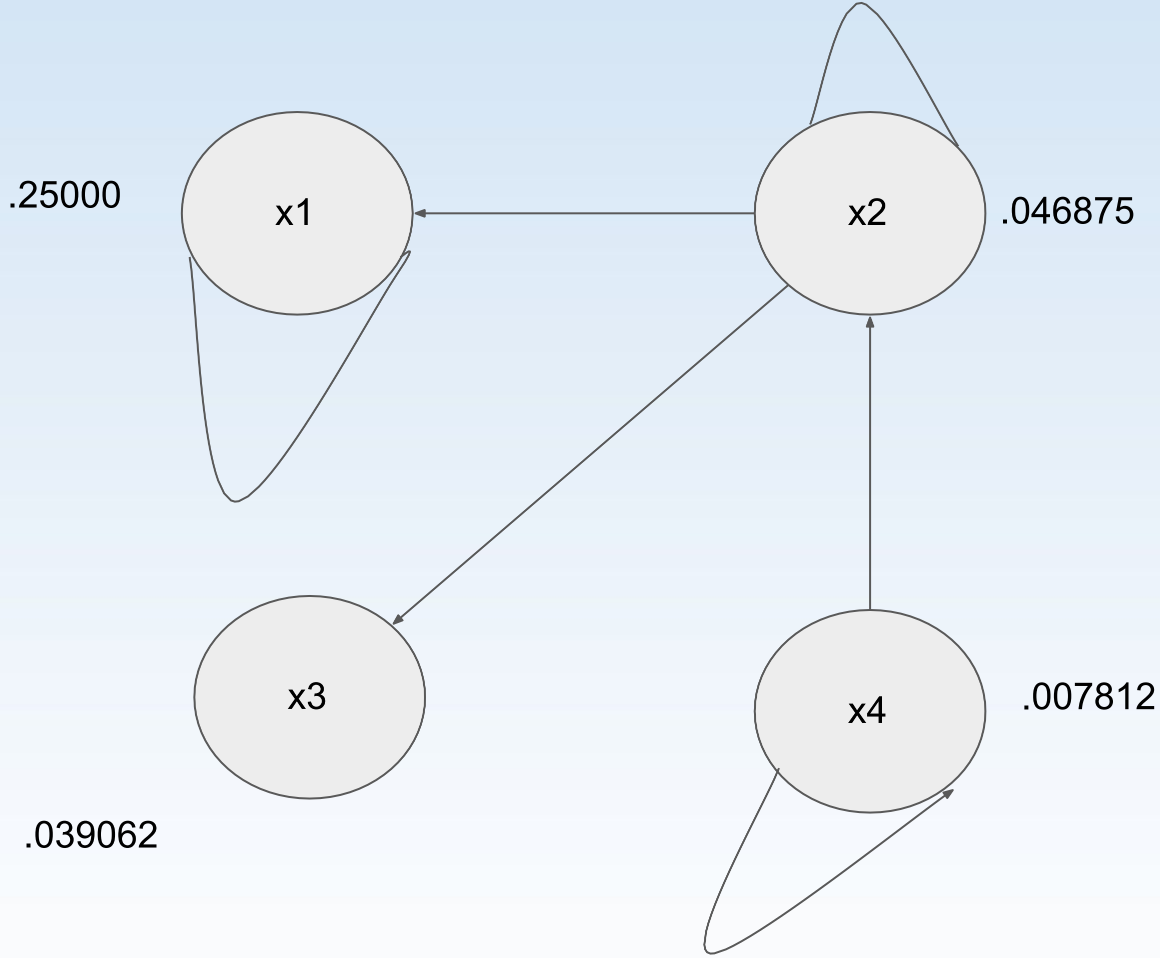

.25000
x1
x2
.25000
x3
x4
.25000
.25000

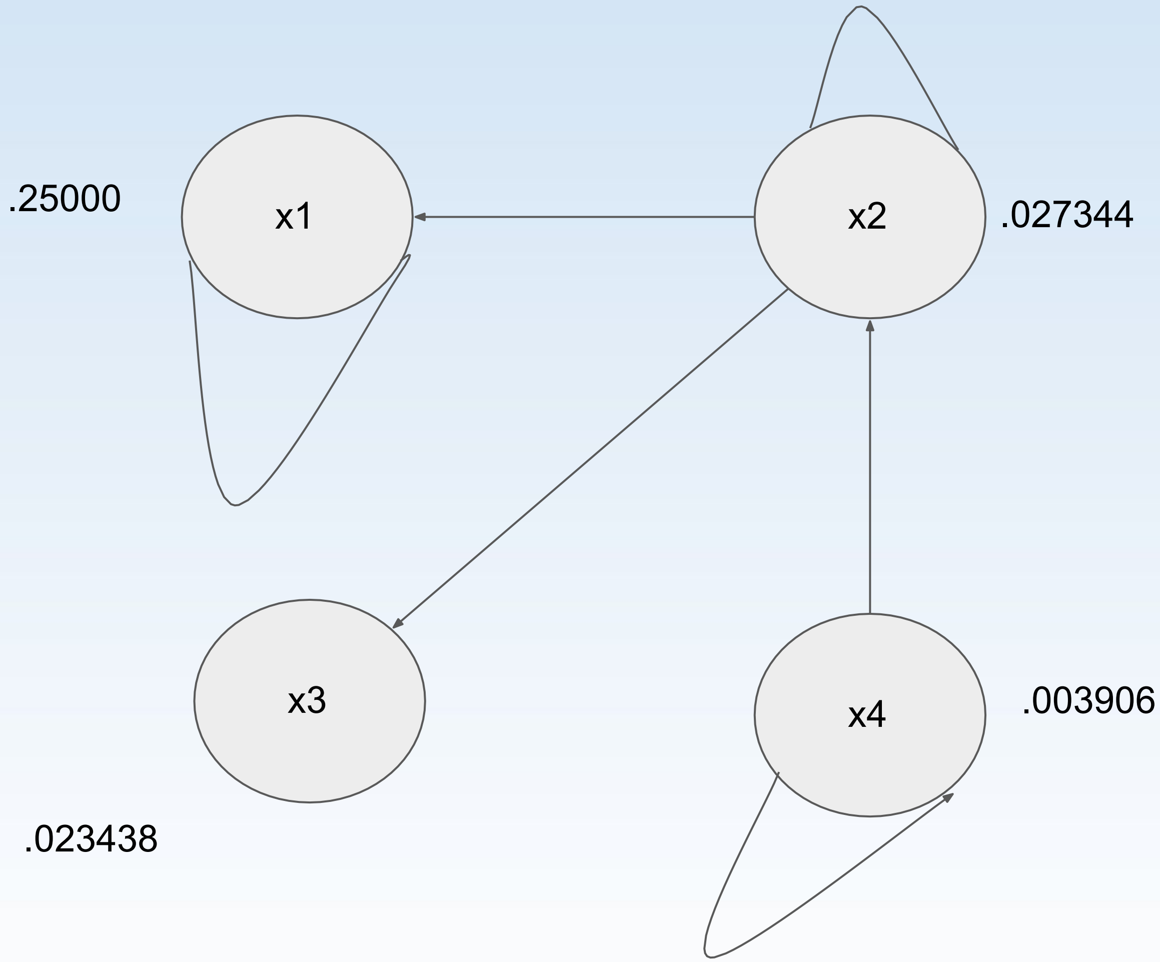

.25000
x1
x2
.25000
x3
x4
.125000
.125000

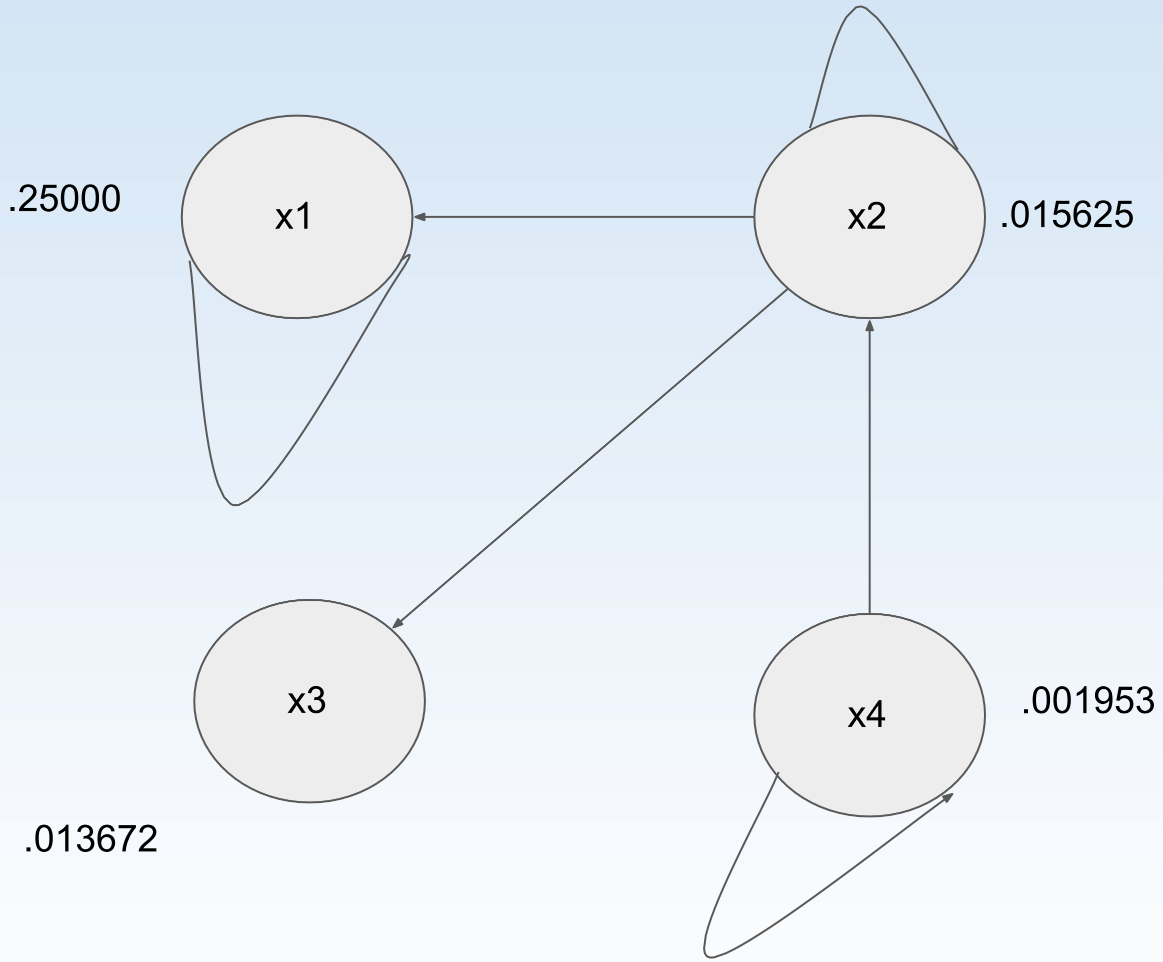

.25000
x1
x2
.187500
x3
x4
.062500
.125000

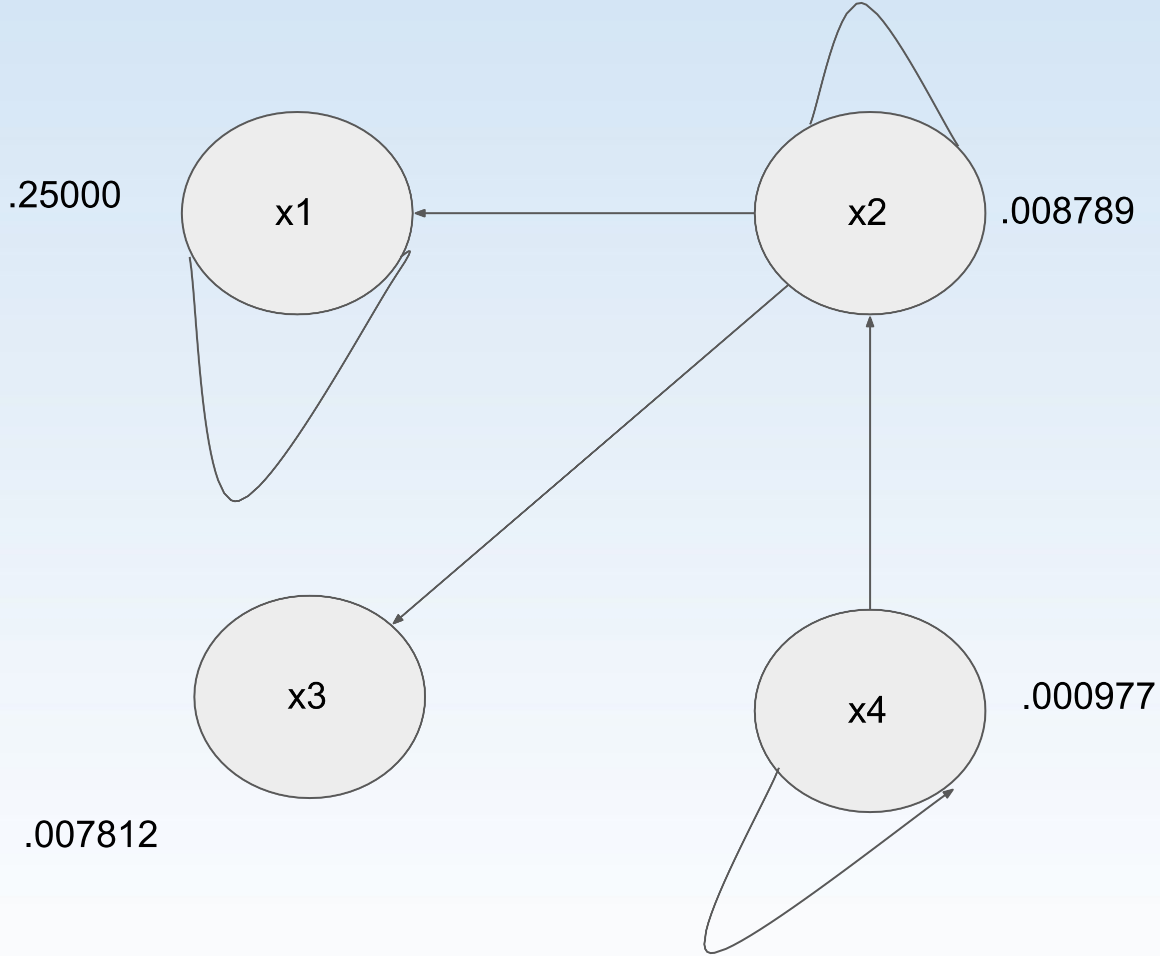

.25000
x1
x2
.125000
x3
x4
.031250
.093750

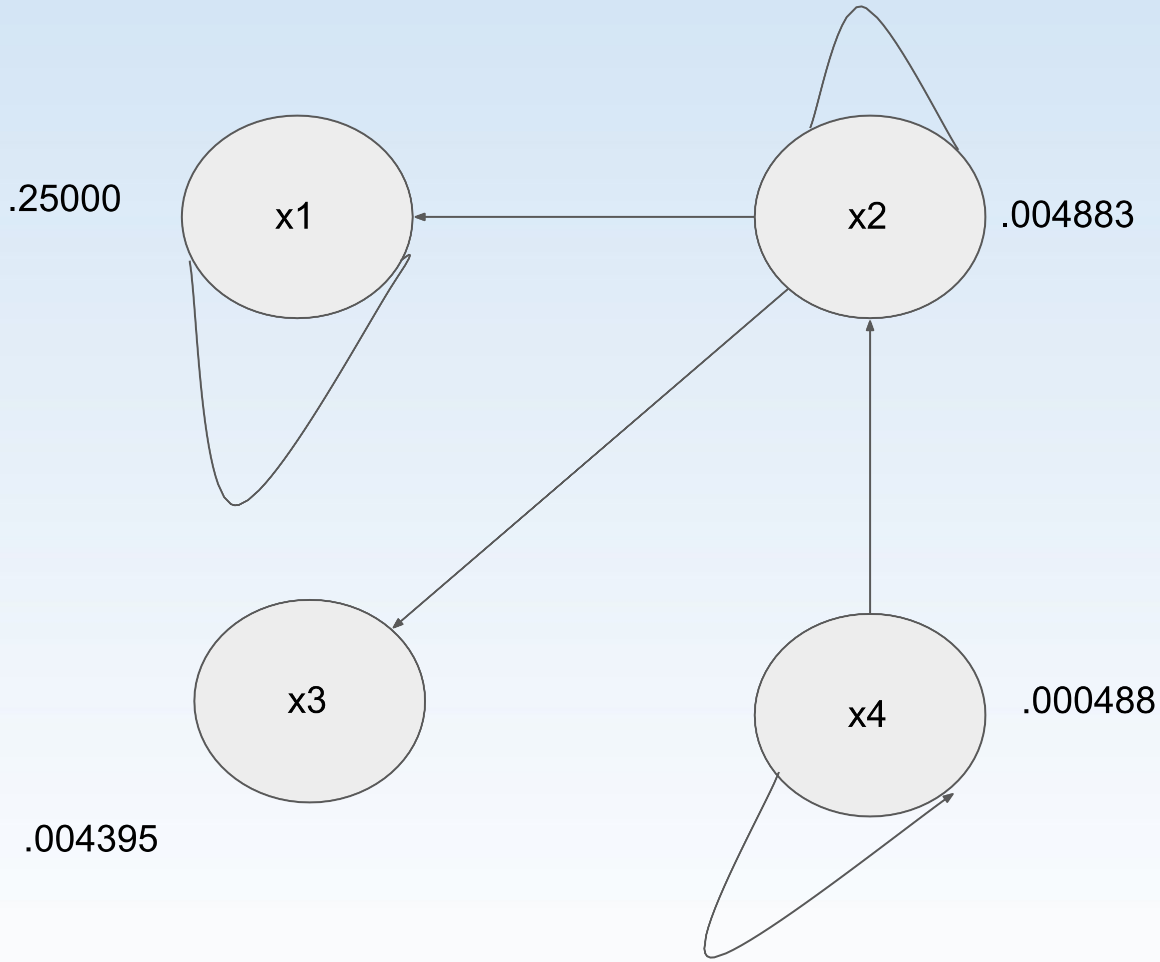

.25000
x1
x2
.078125
x3
x4
.015625
.062500

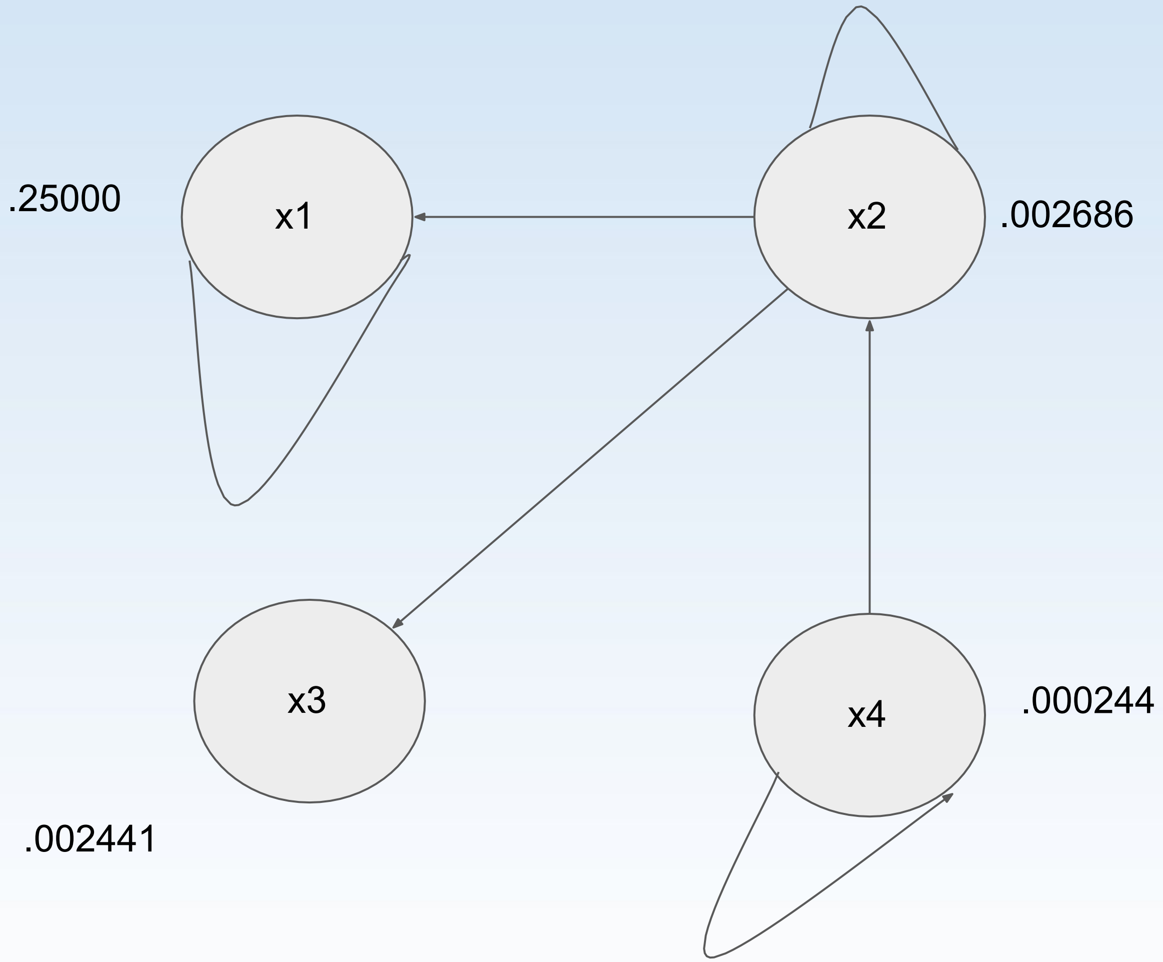

.25000
x1
x2
.046875
x3
x4
.007812
.039062

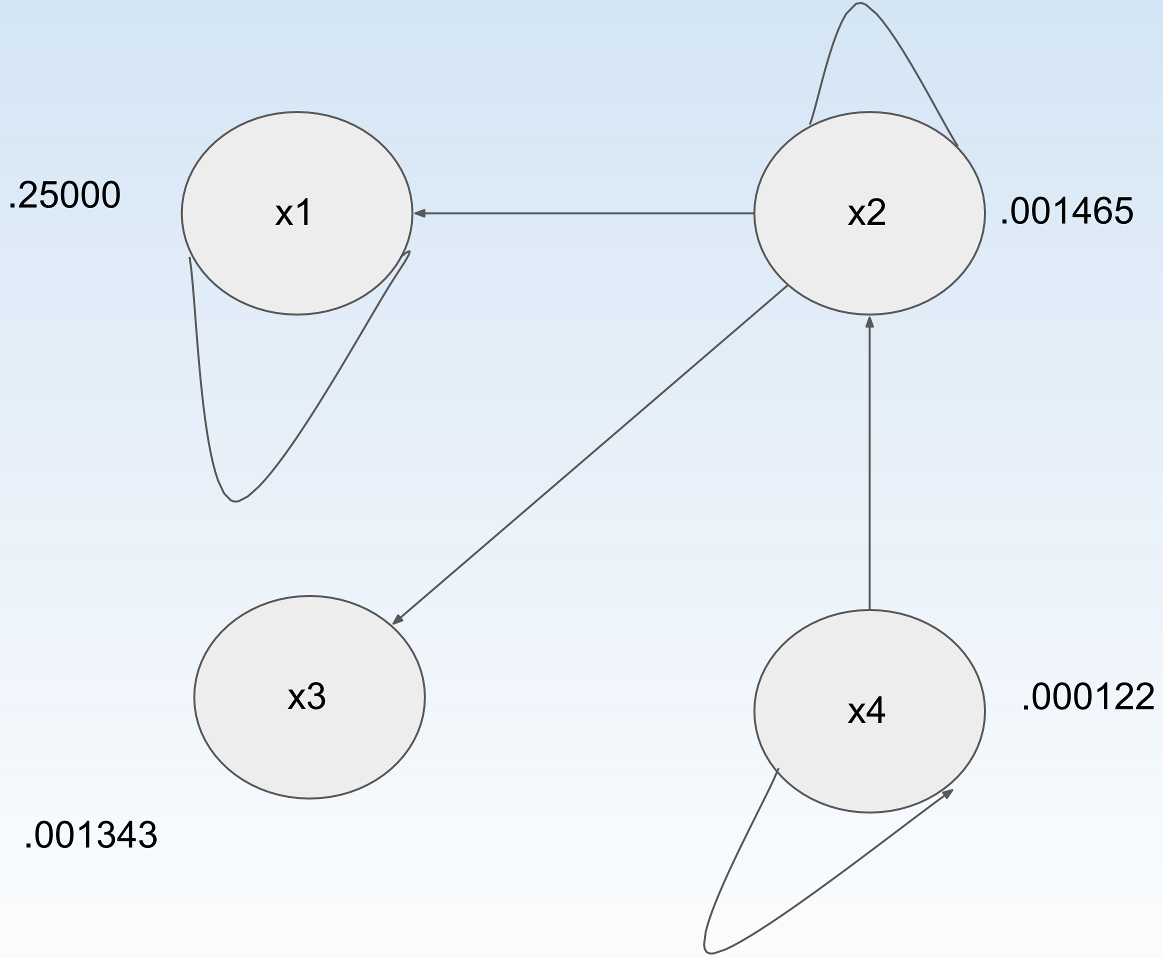

.25000
x1
x2
.027344
x3
x4
.003906
.023438

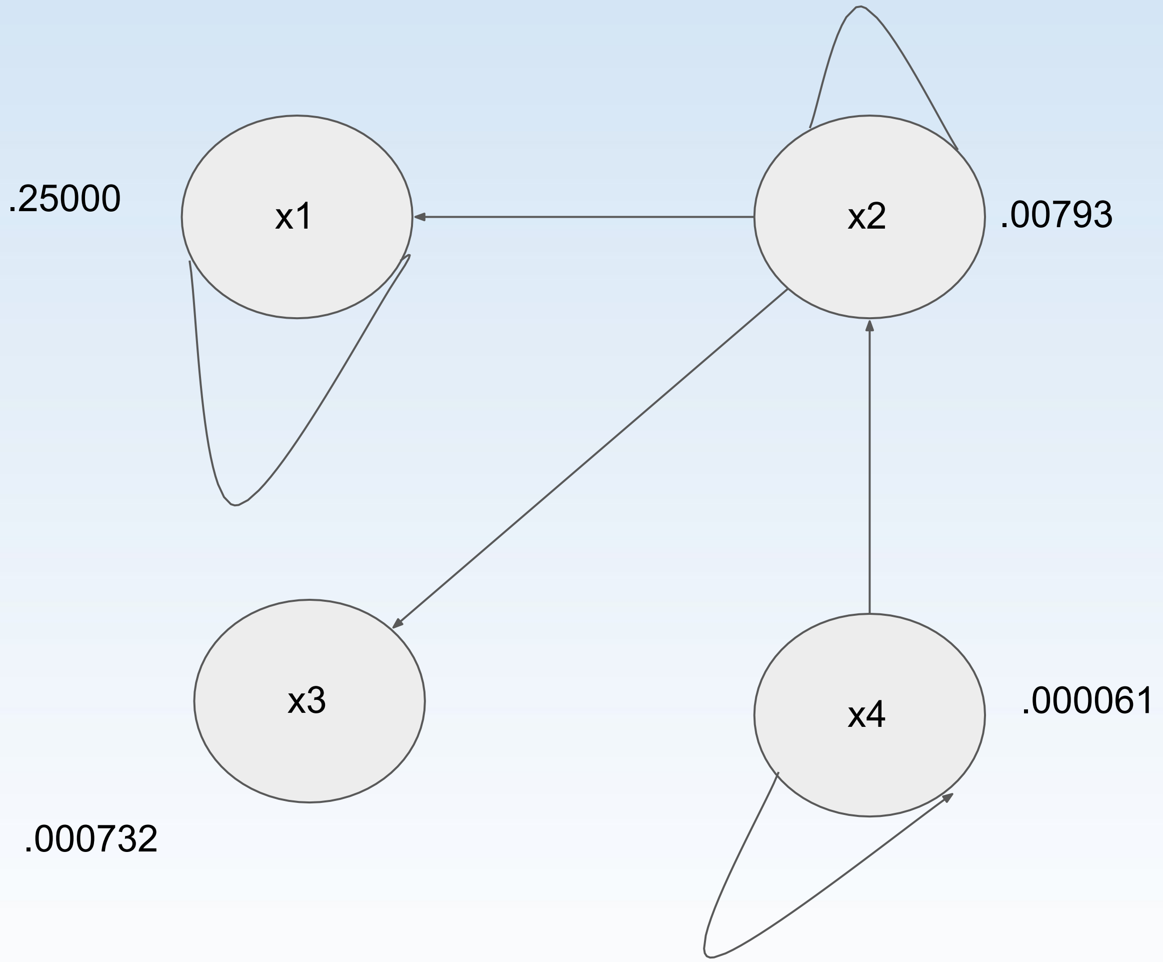

.25000
x1
x2
.015625
x3
x4
.001953
.013672

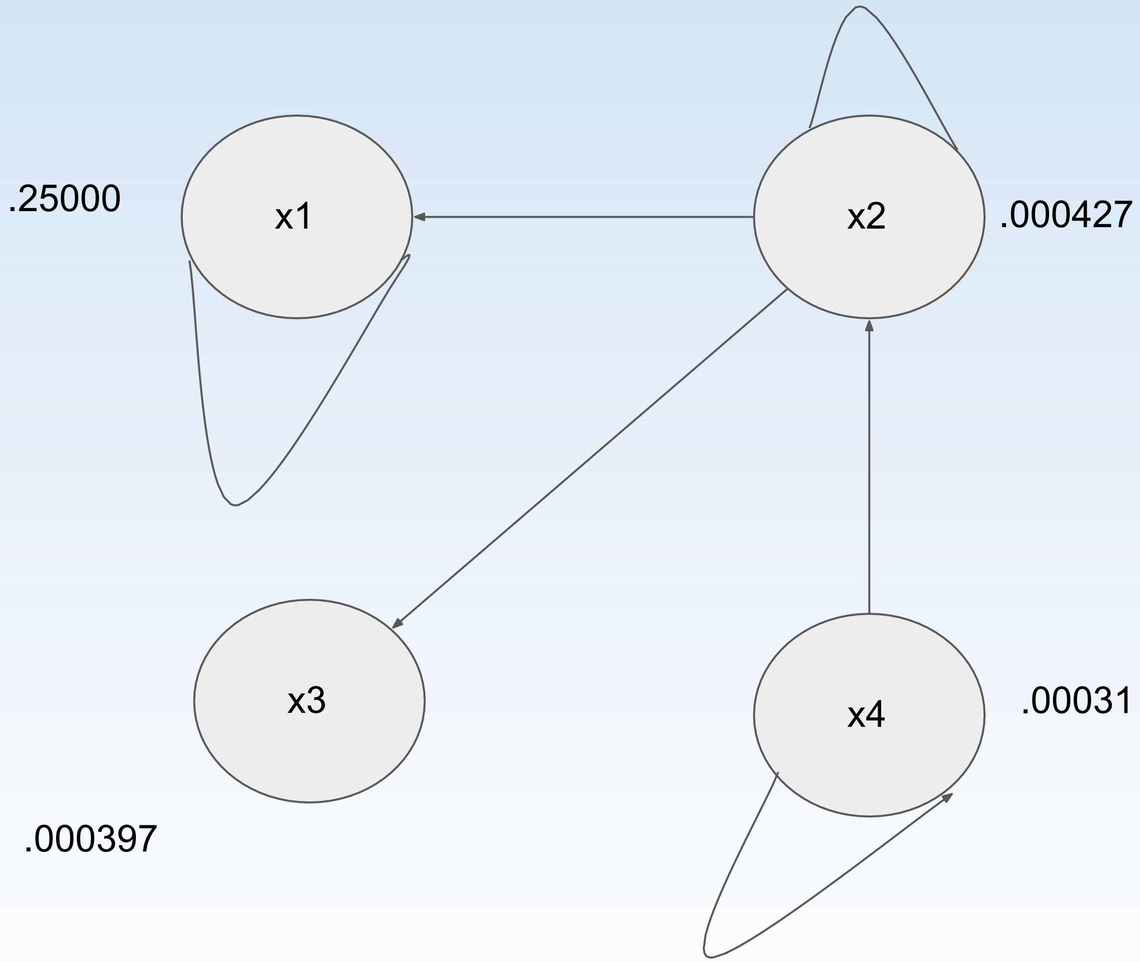

.25000
x1
x2
.008789
x3
x4
.000977
.007812

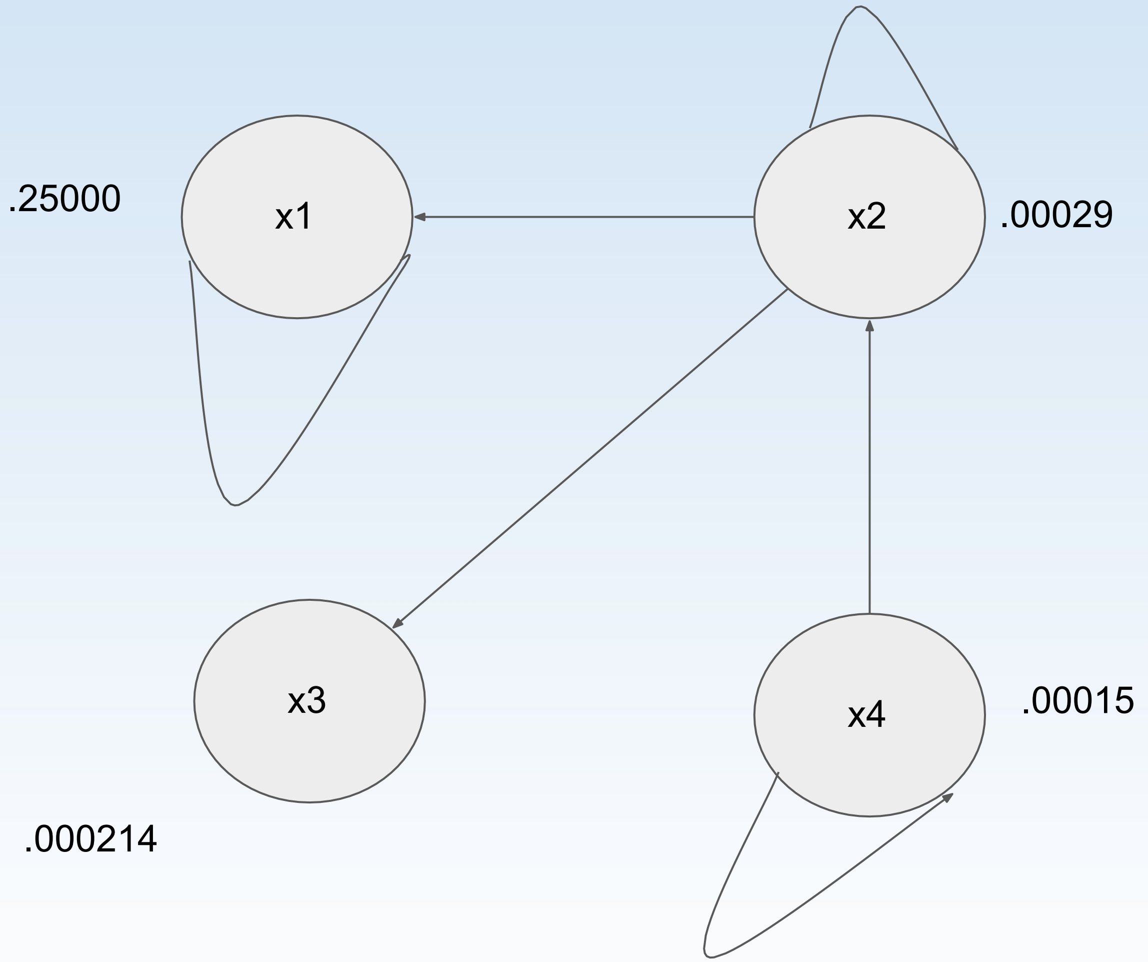

.25000
x1
x2
.004883
x3
x4
.000488
.004395

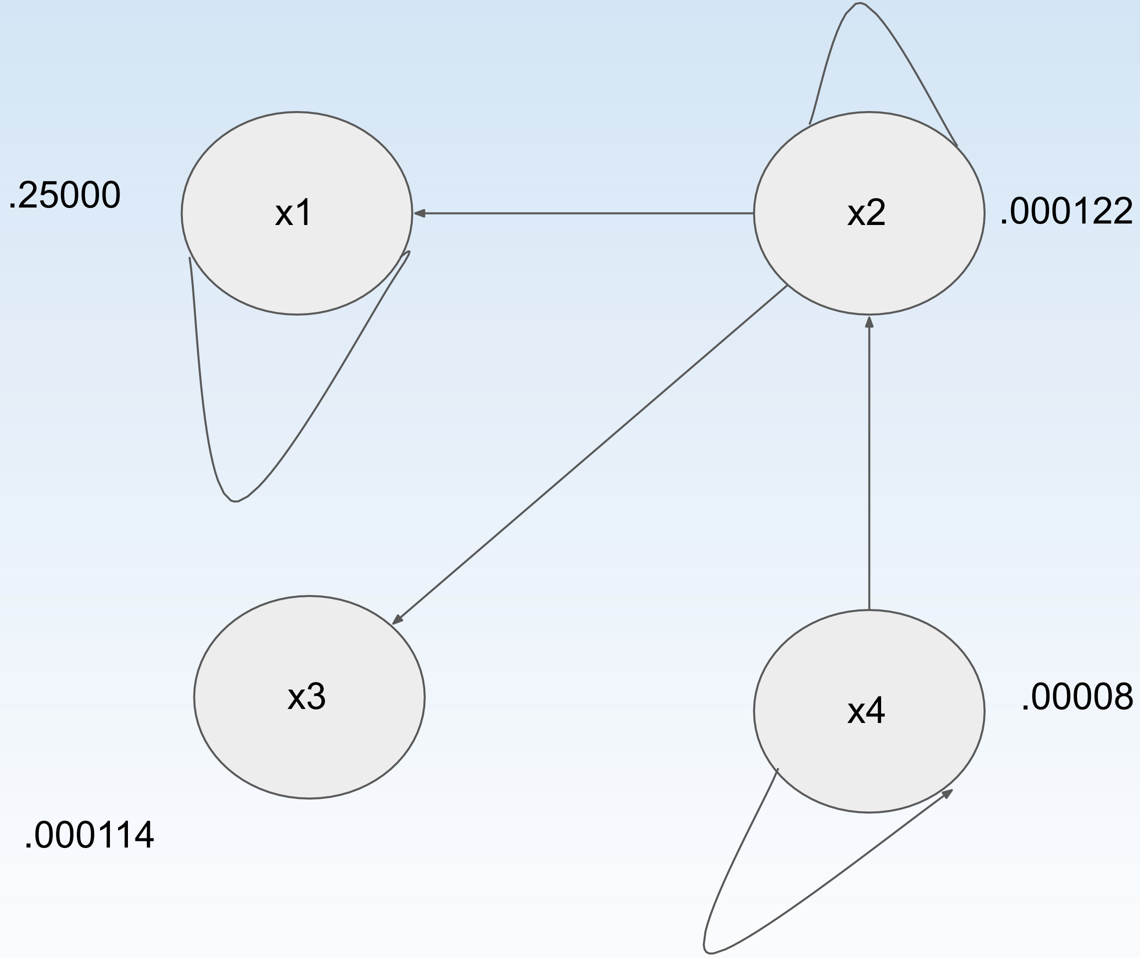

.25000
x1
x2
.002686
x3
x4
.000244
.002441

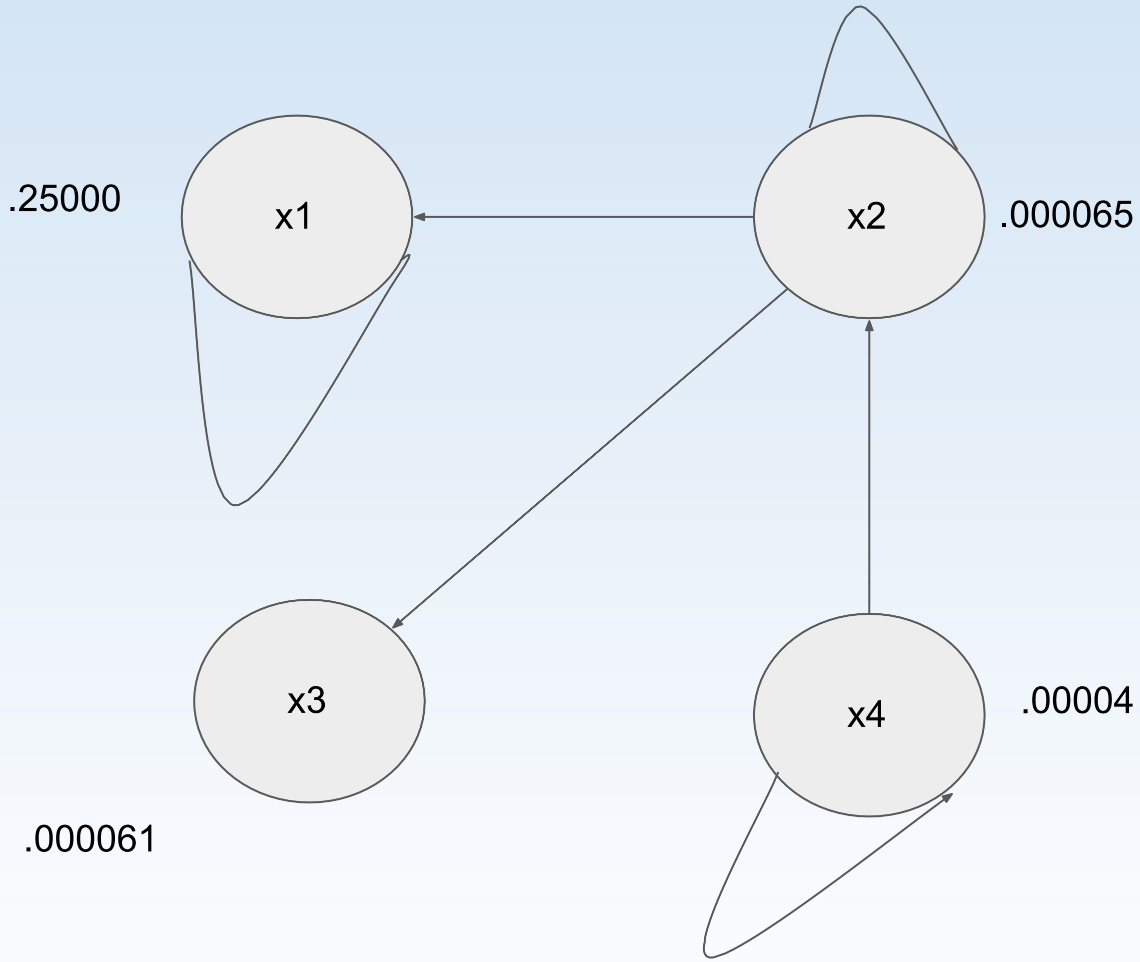

.25000
x1
x2
.001465
x3
x4
.000122
.001343

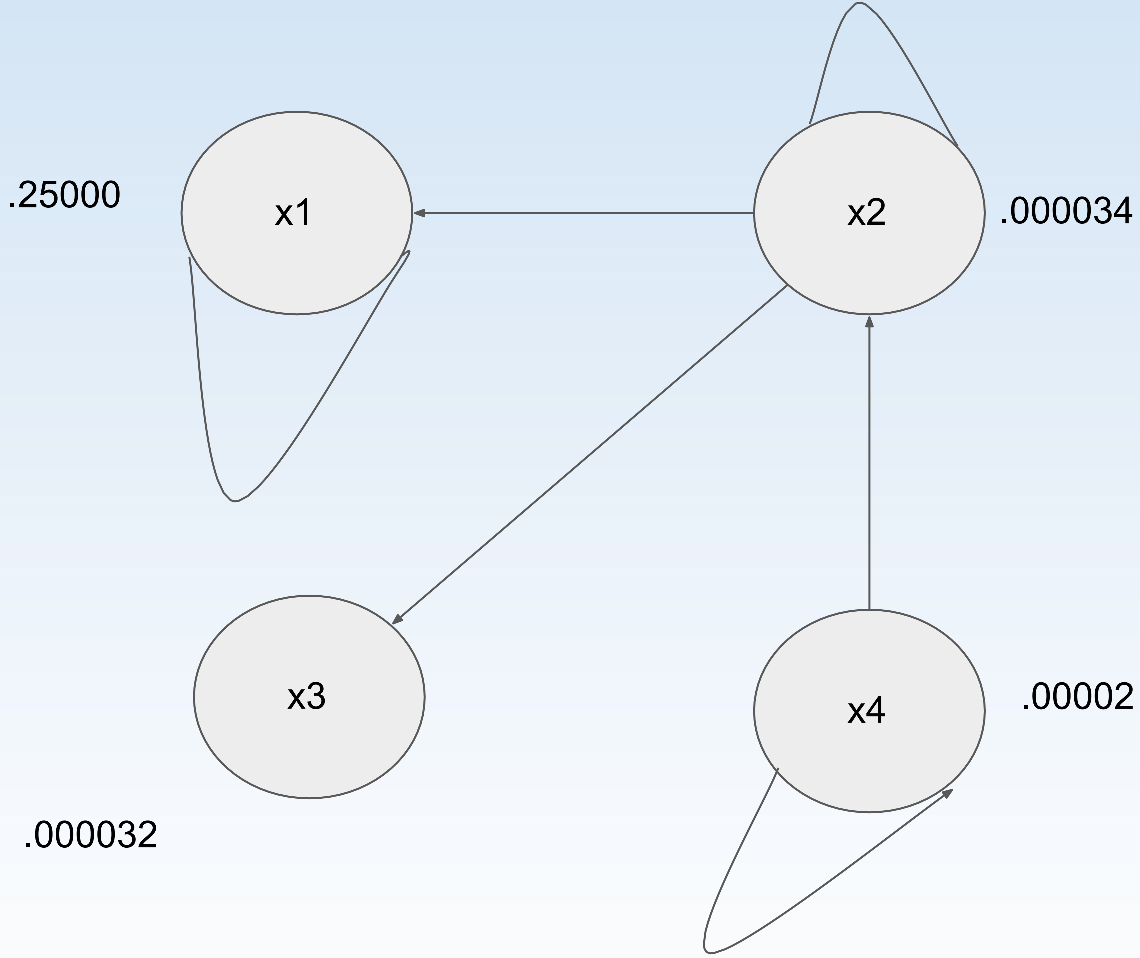

.25000
x1
x2
.00793
x3
x4
.000061
.000732

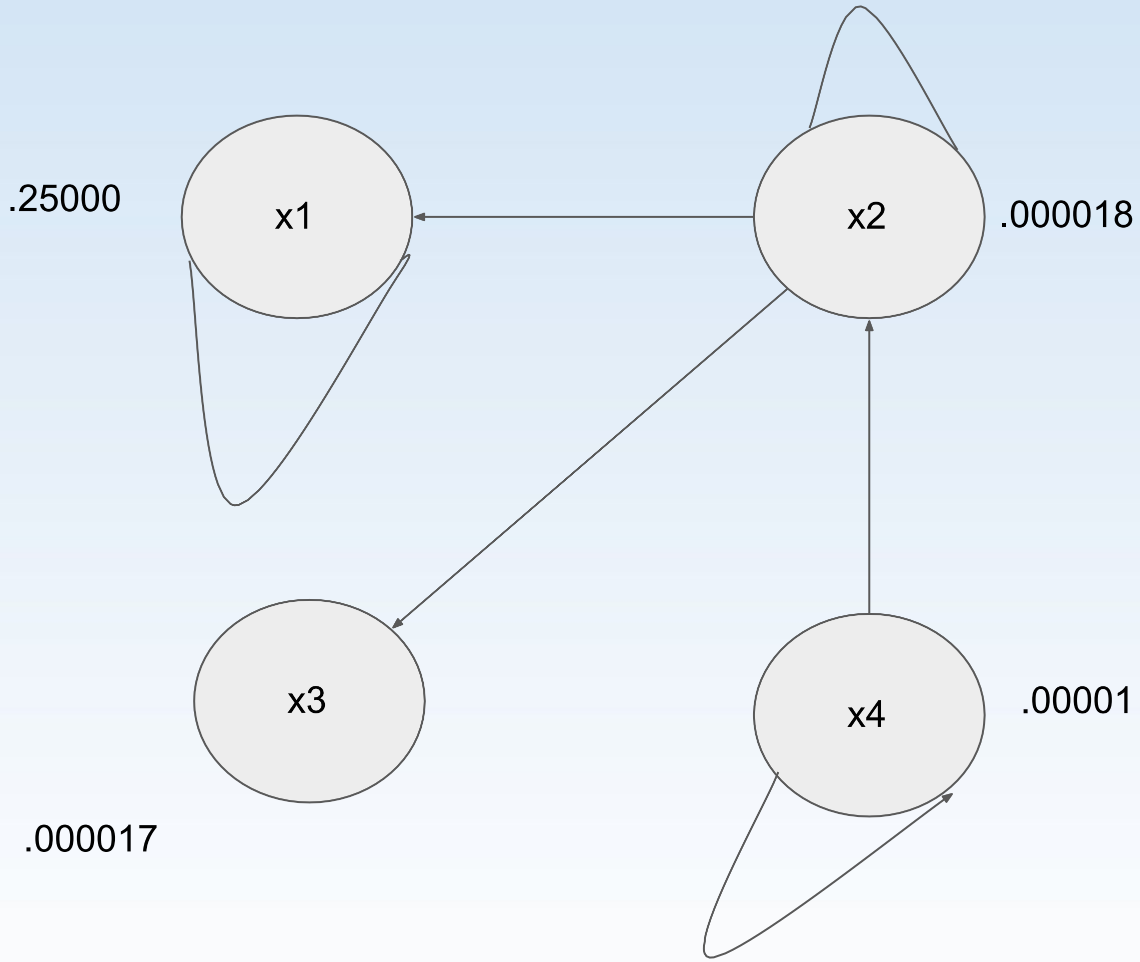

.25000
x1
x2
.000427
x3
x4
.00031
.000397

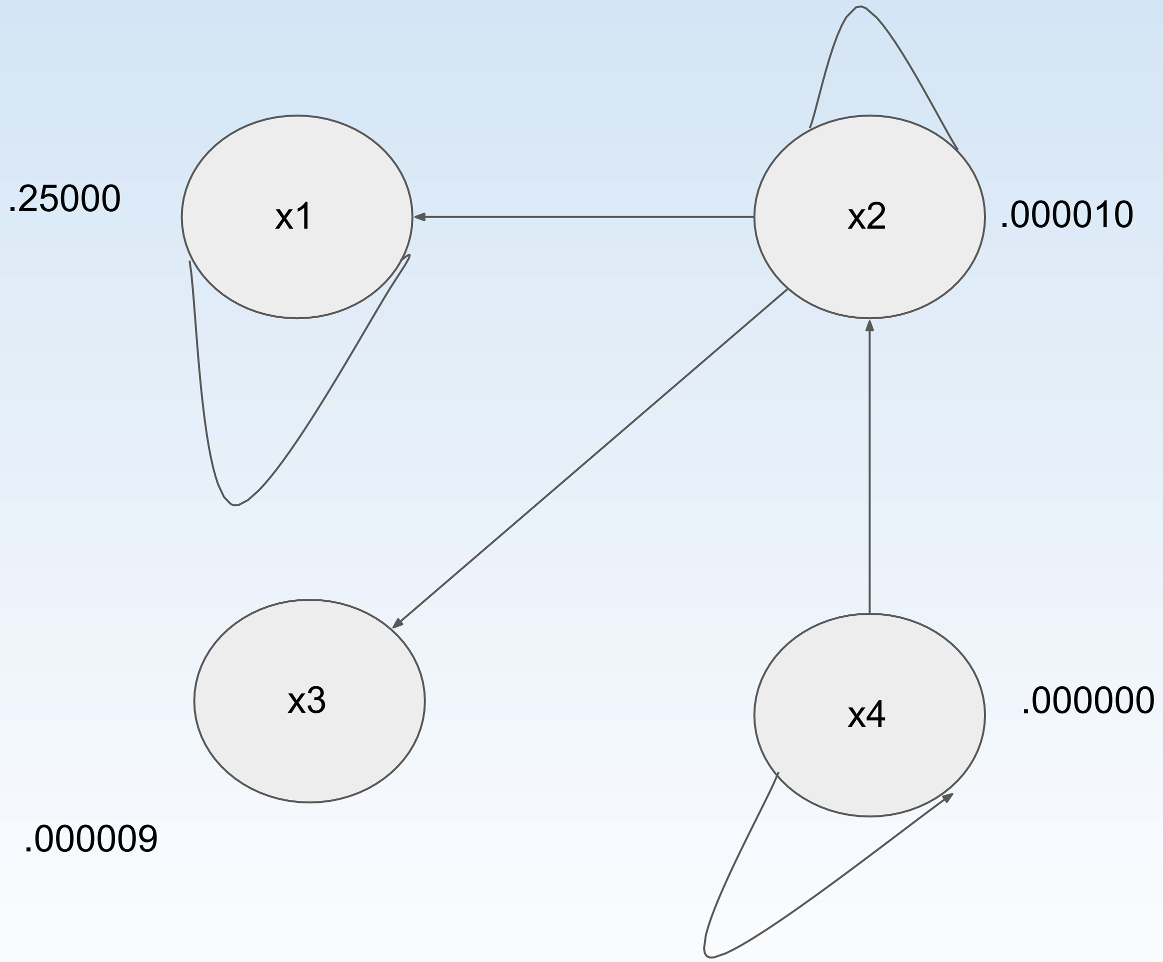

.25000
x1
x2
.00029
x3
x4
.00015
.000214


.25000
x1
x2
.000122
x3
x4
.00008
.000114

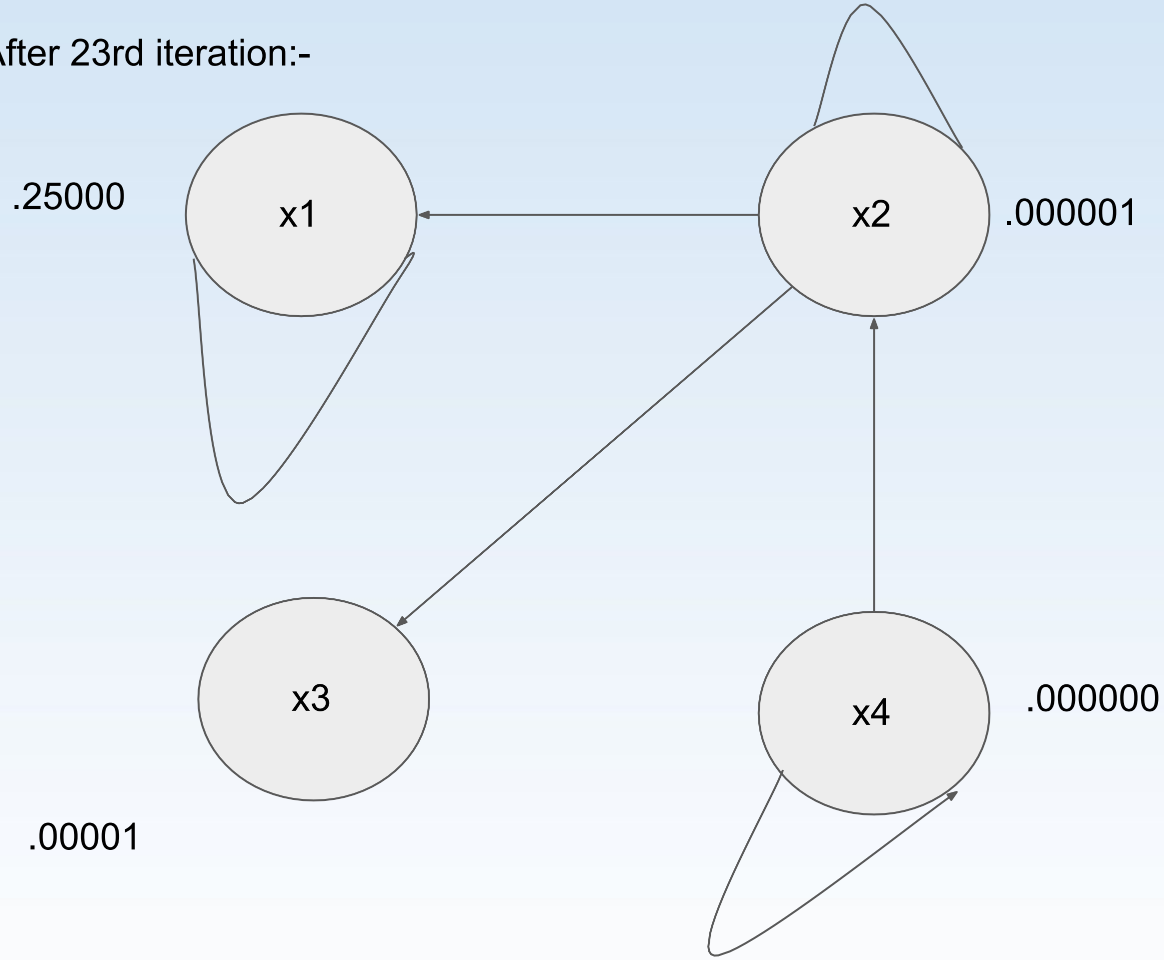

.25000
x1
x2
.000065
x3
x4
.00004
.000061


.25000
x1
x2
.000034
x3
x4
.00002
.000032


.25000
x1
x2
.000018
x3
x4
.00001
.000017


.25000
x1
x2
.000010
x3
x4
.000000
.000009

After 23rd iteration:-



Modified Version of PageRank(Novel PageRank)

PageRank used by Google

$$PR(i) = \frac{(1-d)}{N} + d \cdot \sum_{j \in B_i} \frac{PR(j)}{O_j},$$

N – Total number of webpages
Oj – Number of outgoing links from page j
Bi – Set of web pages pointing to web page i
d – damping factor (usually set to 0.85)

Challenges While computing PageRank

There are many challenges we encounter when calculating PageRank value of web pages:-

- The first difficulty is that the input data is **extremely huge**, therefore, it requires a lot of computing effort.

- The second problem comes from a characteristic of the web i.e it is **dynamic**.

Proposed Solution for the Challenge

Solution for Challenge 1:
1) Parallel approach using , i.e Thread Model Implementation using OpenMp.

2)Approach to provide a parallel solution using GPU-CPU environment that achieves higher accuracy and consumes lesser time to evaluate the PageRank.

Solution for challenge 2 :

Design a parallel approach for dynamic PR computation.

- First, as updates always come in batches, we devise a batch processing method to reduce synchronization cost among every single update and enable more parallelism for iterative parallel execution. And calculate the PR and do the Scaling on the affected nodes.

Dynamic Graph

The importance of nodes in a network constantly fluctuates based on changes in the network structure.
Dynamic graph we denote a graph that is subject to a sequence of updates, such as insertions or deletions of vertices or edges.
Research in graph theory has focused on studying the structure of graphs with the assumption that they are static. But in real scenario, the structure of the graph is dynamic in nature due to which the Pagerank score should be updated after a certain interval of time.

The goal of a our approach is to update the Pagerank after dynamic changes, rather than having to recompute it from scratch each time.
We can classify dynamic graph problems according to the types of updates allowed:-

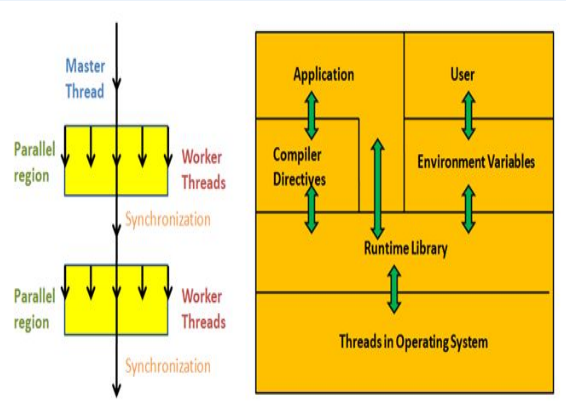
1)**Fully Dynamic** if the update operations include unrestricted insertions and deletions of edges or vertices.
2)**Partially Dynamic** if only one type of update, either insertions or deletions is allowed.
3)**Incremental** only insertions are allowed.
4)**Decremental** only deletions are allowed.

Implementation Architecture

We will now discuss some of the common platforms that are used towards high performance implementations :-

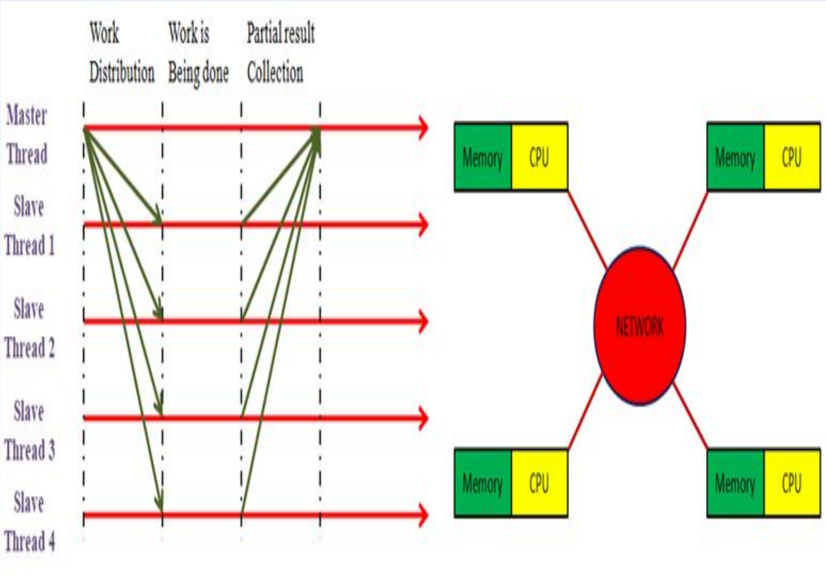
Programming Models : Programming models are conceptual abstractions that allow programmers to implement parallel algorithms.

1) **Thread based model :-** one popular example is the OpenMP library which works via the use of simple compiler directives thus effectively hiding the more complex details of thread creation, termination, and synchronization away from the programmers.



Openmp program execution model and its architecture

2) **Process based models :-** Process based models such as MPI and other models allows accesses only to the private local memories and accesses to remote memories are allowed only via the use of message passing interface.

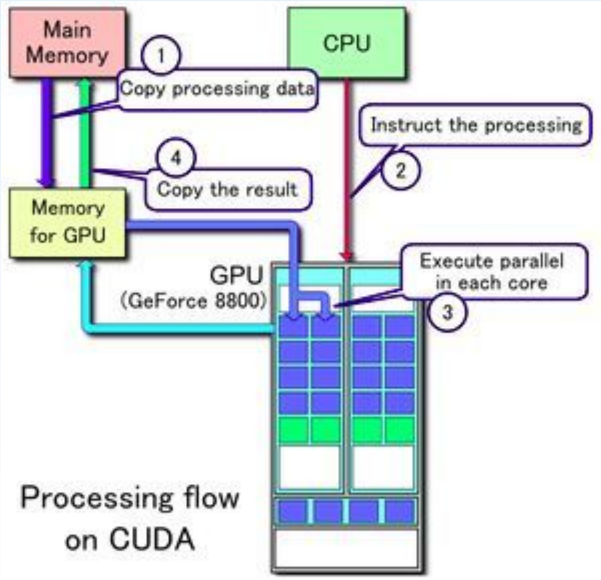


MPI program execution model and its architecture

2) **GPU based model :-** NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

● Serial code executes in a Host (CPU) thread.

● Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements.



CUDA program execution model and its architecture

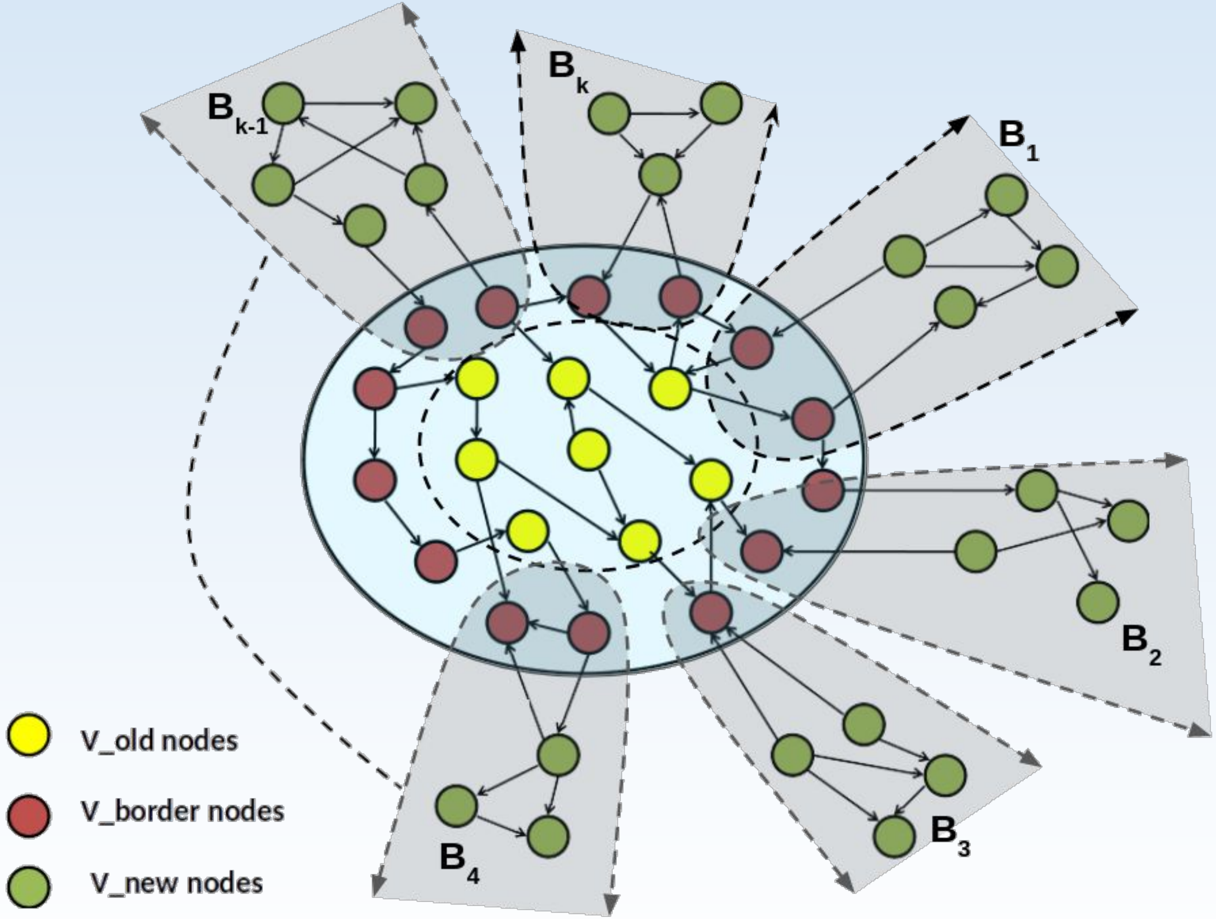
Design Methodologies

The Proposed Solution :

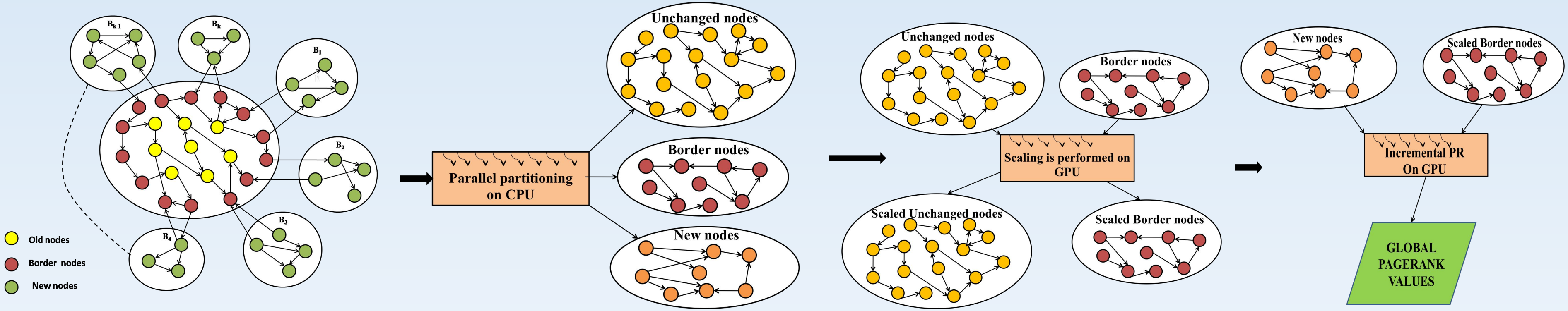
Input : n number of batches containing directed edges(u,v)
Output : Rank of the nodes for every batch update

Phase 1: Calculate the Pagerank of the first batch.
Phase 2 : For the next and previous batch, preprocess the dataset into 3 category:
a) old_node(v_old)
b) border_node(v_border)
c) new_node(v_new)
Phase 3 : Pass the above set to the GPU and do the following:
a) Scaling of older nodes.
b) Scaling of border nodes.
c) PageRank computation of the newer nodes.
Phase 4 : Collect all the ranks of the nodes and merge.

Identification of nodes



Hybrid PageRank computation



a b c d

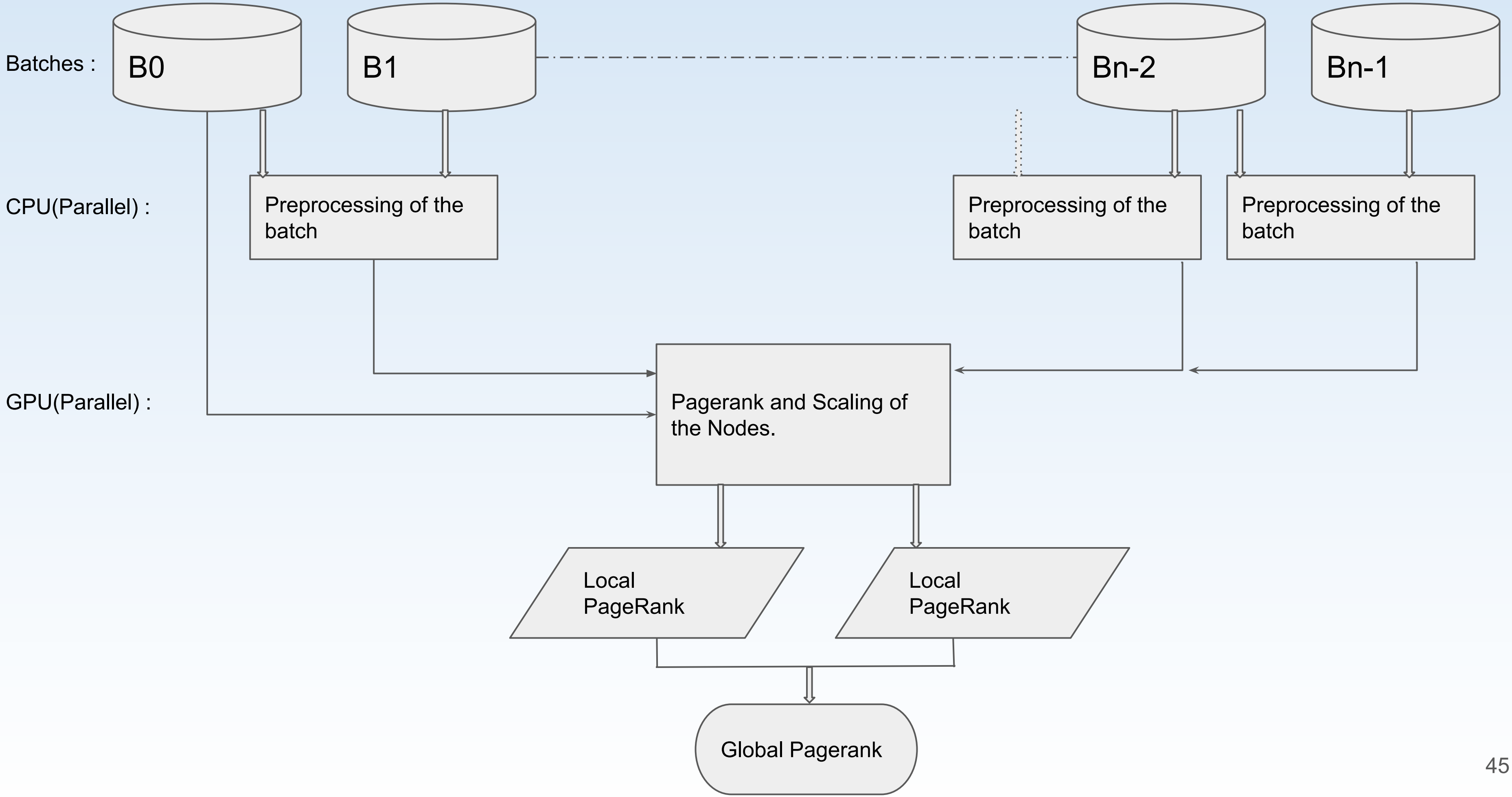
(a) Graph with batch updates

(b) Identification of v_border, v_new and v_old nodes

(c) Scaling on v_old and v_border nodes

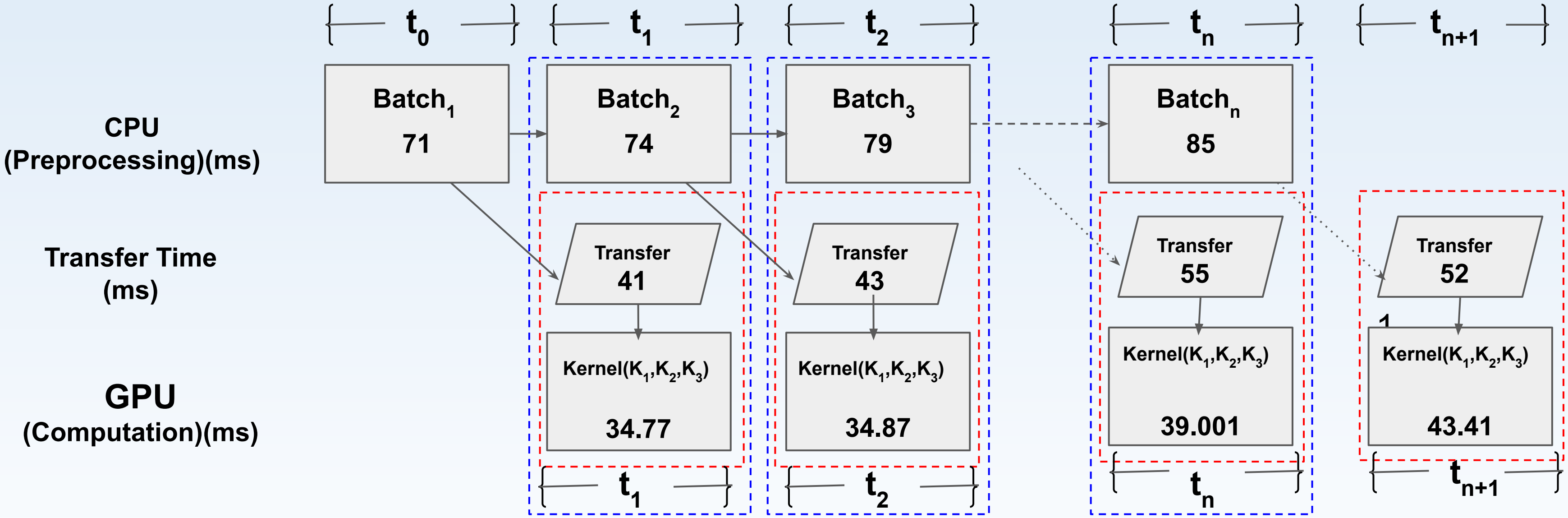
(d) PageRank Computation on v_new nodes

Overview :



Batches :
B0
B1
Bn-2
Bn-1
CPU(Parallel) :
Preprocessing of the batch
Preprocessing of the batch
Preprocessing of the batch
GPU(Parallel) :
Pagerank and Scaling of the Nodes.
Local PageRank
Local PageRank
Global Pagerank

Overlapping between CPU PreProcessing and GPU Processing



Hybrid Incremental PageRank Computation Algorithm

I/P: n number of batches containing directed edges(u,v)
O/P: Rank of the node

Phase 1:
GPU :: Calculate the Pagerank of the first batch.

Phase 2: For the next and previous batch, preprocess the dataset into 3 category:-

1) v_old
2) v_new
3) v_border

```
Batches=[B0,B1,B2..................Bn]-B0
```
assign 3 threads to CPU
CPU ::
```
for batch in Batches do:
    v_old=set()
    v_new=set()
    v_border=set()
    v_temp=set()
    for edge in batch.edges()
                v_new.add(edge)
    endfor
    for edge in B0.edges()
                v_old.add(edges)
    endfor
    while(v_new.size !=0)
                n=v_new.pop()
                if(n there in in v_temp) continue
                endif
                v_temp.add(n)
                for i in batch.successor(n)
                            v_old.delete(i)
                            v_new.add(i)
                endfor
    endwhile
    for i in v_temp:
                for i in batch.predecessor(i)
                            b_border.add(i)
                endfor
    endfor
    v_border=v_border-v_temp
endfor
```
CPU->GPU :: Transfer all the 3 set of vertex to the GPU.

Phase 3: Calculate the Pagerank of the new nodes and border nodes and do scaling to the remaining ones.
assign one thread to GPU
GPU ::
```
for i in v_old parallel do:
    dest[i]=no_of_nodes/v_old.size()*rank[i]
endfor
for i in v_border parallel do:
    dest[i]=no_of_nodes/v_border.size()*rank[i]
endfor
for i in v_new parallel do:
    dest[i]=(1-damping_factor)/no_of_nodes+(damping_factor*sum_ranks)
endfor
```

Phase 4:
GPU->CPU :: Transfer all the ranks of the nodes

Algorithms :

Algorithm 2: Basic hybrid PageRank computation

Require: Scratch graph G and k number of batches in B containing directed edges(u,v,flag) as well as initialized PageRank vector dest.

Ensure: Rank of the nodes in vector dest

{**Phase 1: Pre-processing phase**}

CPU::Partition the incoming batches based on insertion and deletion

$(v\_old, v_border, v_new, v_target)$=createPartition$(G,B)$

{**Phase 2: PageRank updation**}

INSERTION: Generate threads equal to number of v_old, v_border, v_new

for $\forall u\varepsilon v\_old$ in parallel **do**

    GPU:: $dest[u] = \frac{|G|*dest[u]}{|v\_old|}$ {Scaling}

end for

for $\forall x\varepsilon v\_border$ in parallel **do**

    GPU:: $dest[x] = \frac{|G|*dest[u]}{|v\_border|}$ {Scaling}

end for

for $\forall y\varepsilon v\_new$ in parallel **do**

    GPU:: $dest[y] = \frac{1-d}{G} + (d * sum\_ranks)$ {PR Update}

end for

for $\forall z\varepsilon v\_border$ in parallel **do**

    GPU:: $dest[z] = \frac{1-d}{G} + (d * sum\_ranks)$ {PR Update}

end for

DELETION: Generate threads equal to number of v_old and v_target

for $\forall u\varepsilon v\_target$ in parallel **do**

    GPU:: $dest[u] = \frac{1-d}{G} + (d * sum\_ranks)${PR Update}

end for

for $\forall v\varepsilon v\_old$ in parallel **do**

    GPU:: $dest[v] = \frac{|G|*dest[u]}{|v\_old|}$ {Scaling}

end for

Algorithm 3: createPartition(G,B)

Require: Scrath graph G and k number of batches in B containing directed edges(u,v) having corresponding insert or delete update

Ensure: v_old, v_new, v_border, v_target

{**For insert update**}

CPU :: Generate threads using OpenMP

for $\forall b_i \varepsilon B$ in parallel : **do**

    Initialize $v\_old, v\_new, v\_border, v\_temp, v\_target$

    Push $\forall(u,v)\varepsilon batch$ to v_new

    Push $\forall u\varepsilon G$ to v_old

    while $(v\_new! = NULL)$ **do**

        Pop element $x\varepsilon v\_new$

        if (x ϵv_temp) **then**

            Continue

        end if

        Push x to v_temp

        for every successor y of $x\varepsilon G$ **do**

            Push y to v_temp

        end for

    end while

    for all $z\varepsilon v\_temp$ **do**

        for if there is a predecessor l present in incoming batch **do**

            Push l into v_border

        end for

    end for

    {**For delete update**}

    for $\forall u, v\varepsilon$ G **do**

        Pop element u and v from G

    end for

    for $\forall y$, successor of u,v ε G **do**

        Push y into v_target

    end for

end for

return v_old,v_new,v_border,v_target

Experimental Results and Observation

Experimental Setup: We tested our implementations for PR both for Serial and Parallel approach

CPU Info :
Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
Frequency : 2.10 GHz
CPU(s) : 32
Thread(s) per core : 2
Core(s) per socket : 8

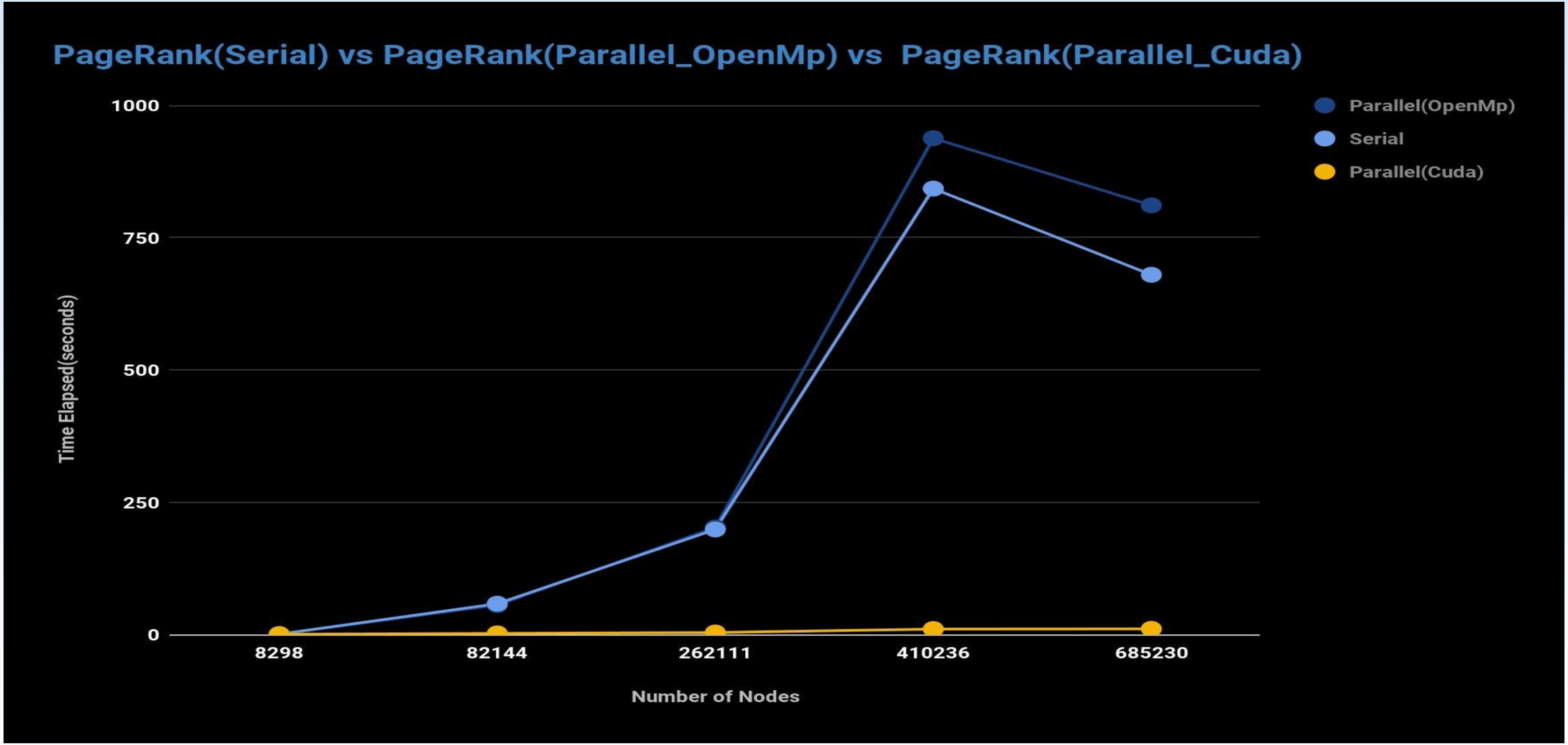
GPU Info :
NVIDIA Corporation GV100GL [Tesla V100 PCIe 32GB]
clock: 33MHz
width: 64 bits

Memory Info
Total RAM: 128 GB

DataSets Used for Computing PageRank

Dataset	|V|	|E|
Youtube[7]	1.10 M	2.90 M
Amazon[7]	0.41 M	3.35 M
web-Google[7]	0.87 M	5.10 M
wiki-Topcat[7]	1.79 M	28.51 M
soc-pokec[7]	1.63 M	30.62 M
Reddit[7]	2.61 M	34.40 M
soc-LiveJournal[7]	4.84 M	68.99 M
Orkut[7]	3.00 M	117.10 M
Graph500[7]	1.00 M	200.00 M
Random[7]	1.00 M	200.00 M
NLP[7]	16.24 M	232.23 M
Arabic[7]	22.74 M	639.99 M

Experiment for Challenge 1

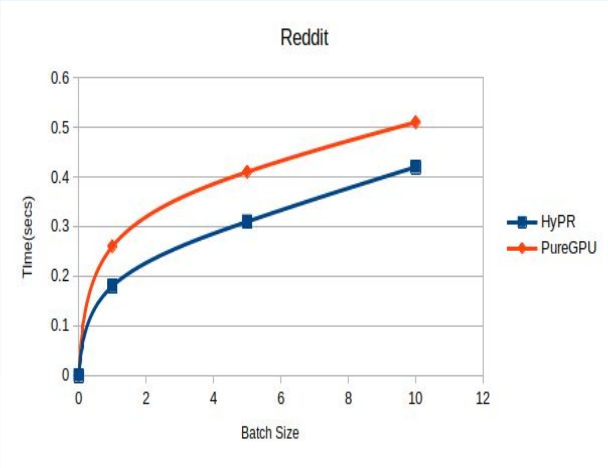


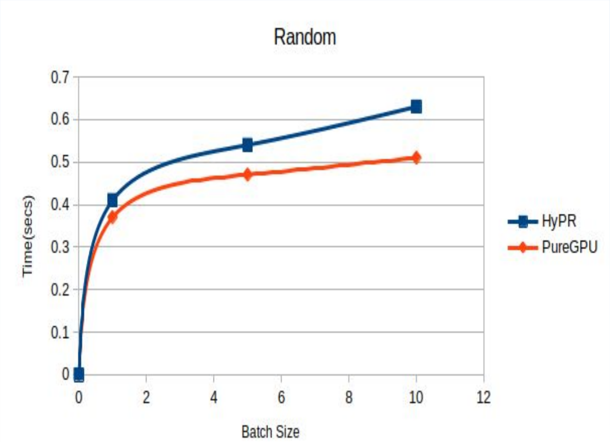
Analysis(OpenMp vs Serial vs Cuda)

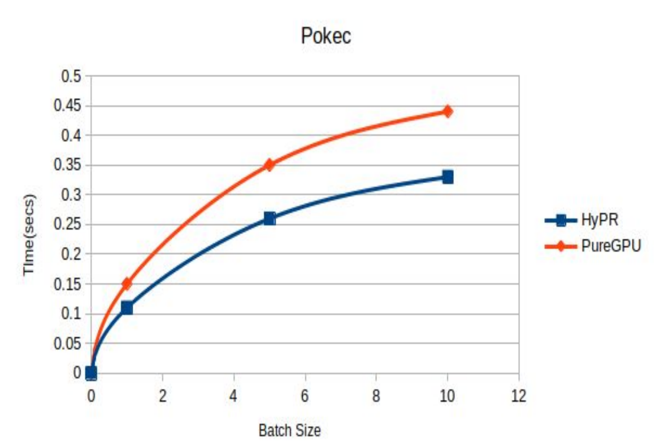
Experiment for Challenge 2

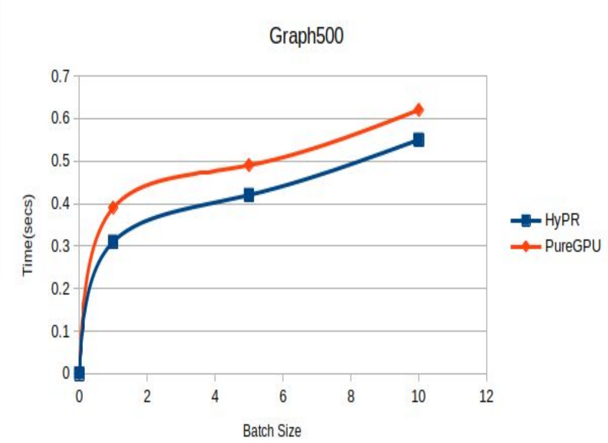
Update time

In this experiment, we conducted the experiments over two approach. One with the Hybrid(HyPR) and the other with PureGPU using the same algorithm with different batch sizes, i.e batch with 1%, 5%, 10% of the edges of the data-set we used respectively. From the given figure we found that the time computation to compute the Page-Rank by Purely GPU based over the Hybrid approach shows a lower performance.



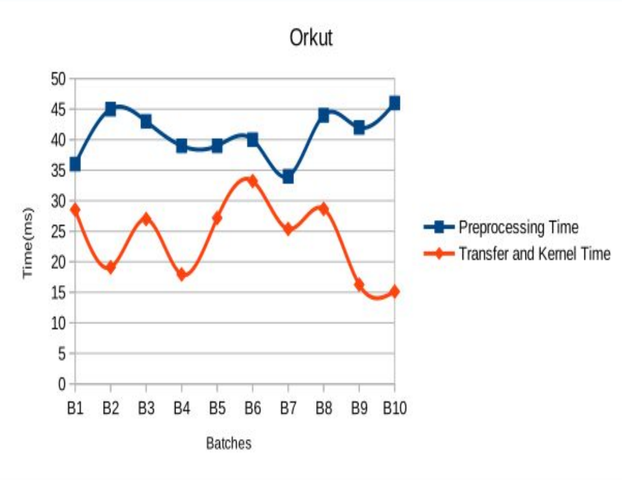


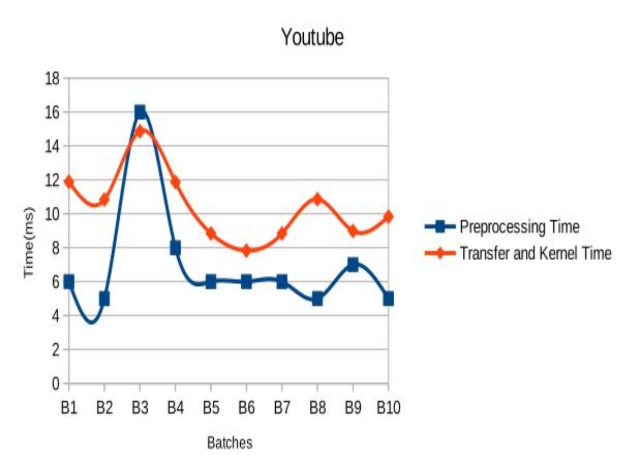


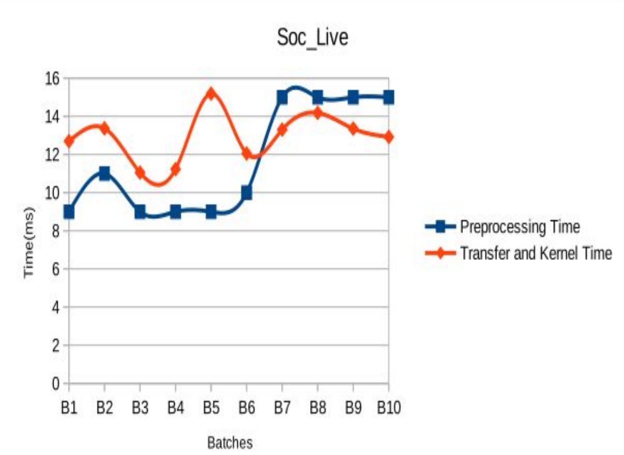


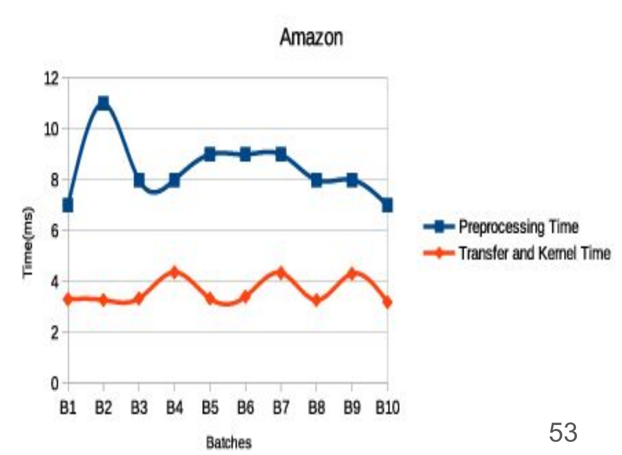
Hybrid Overlaps

Our motive to do this experiment is to analyse how much GPU has to wait for the CPU to finish its preprocessing in order to the rest of the computation task. On experimenting we found that number of threads was directly proportional to the lesser waiting time by the GPU, i.e more the number of threads, lesser the waiting time. This experiment was done with the 16 threads for doing preprocessing and achieved almost an overlap of the preprocessing time with kernel time. So with our proposed approach with eight graph data-sets and we find that time computation of the pre-processing and of kernel is almost overlapped.

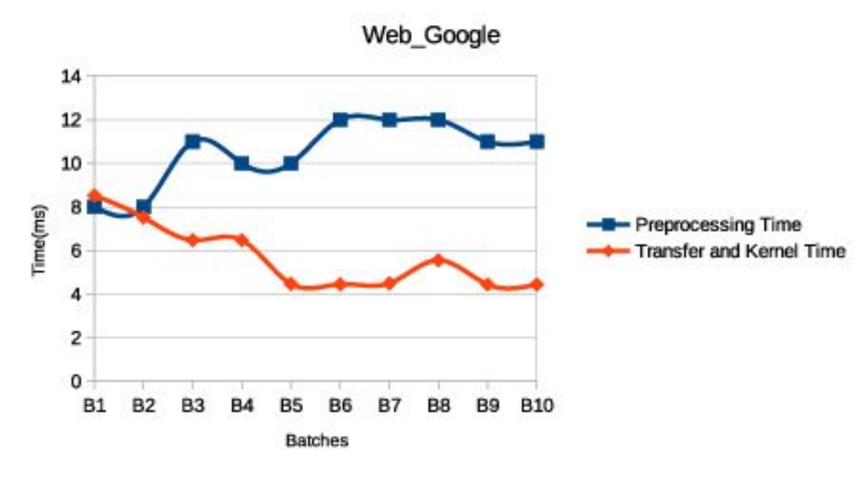




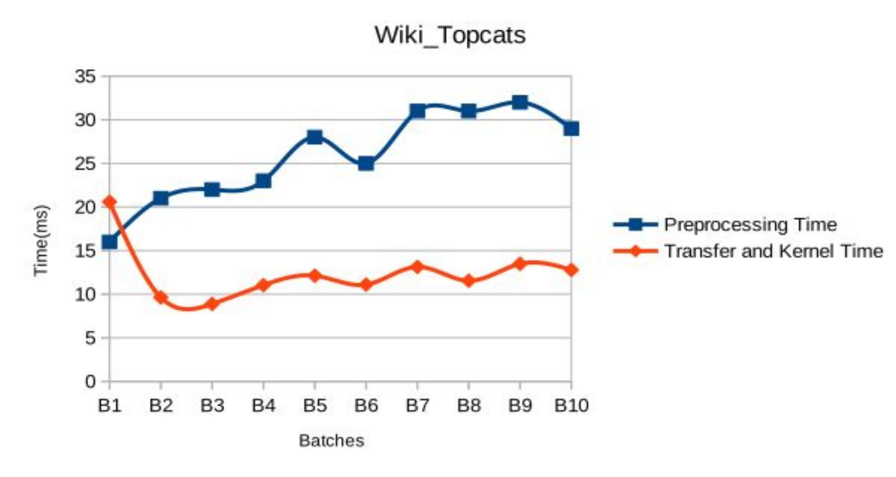




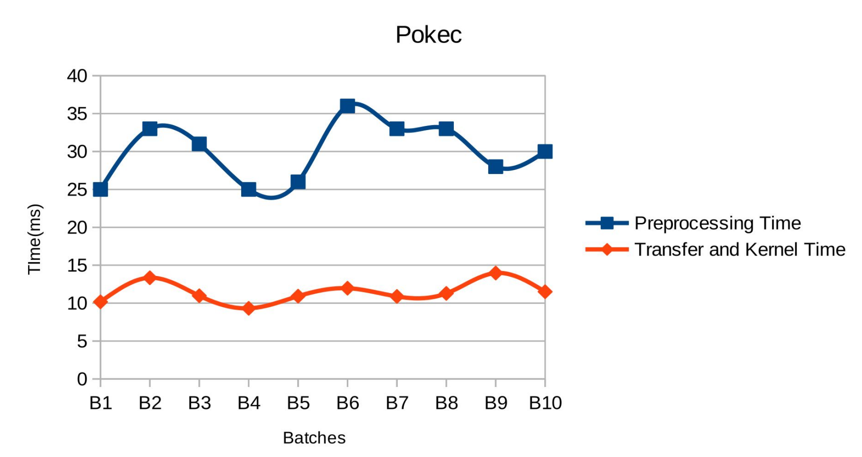
Contin...



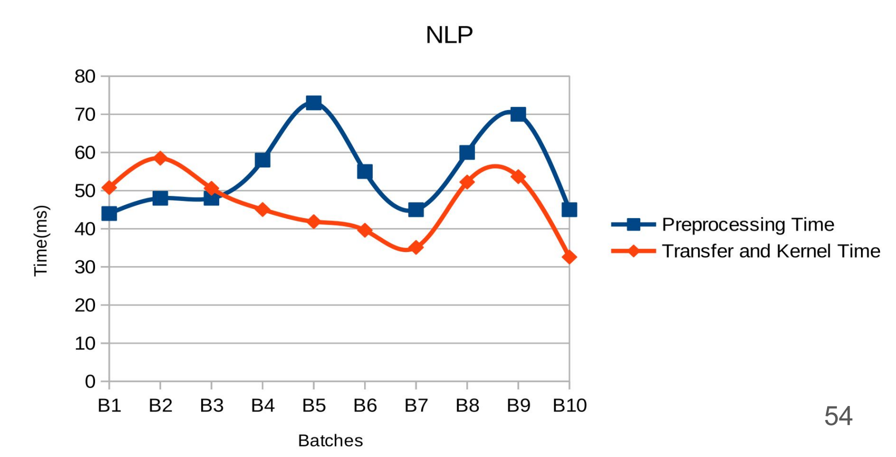
Web_Google
14
12
10
8
6
4
2
0
Time(ms)
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10
Batches
Preprocessing Time
Transfer and Kernel Time



Wiki_Topcats
35
30
25
20
15
10
5
0
Time(ms)
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10
Batches
Preprocessing Time
Transfer and Kernel Time



Pokec
40
35
30
25
20
15
10
5
0
Time(ms)
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10
Batches
Preprocessing Time
Transfer and Kernel Time



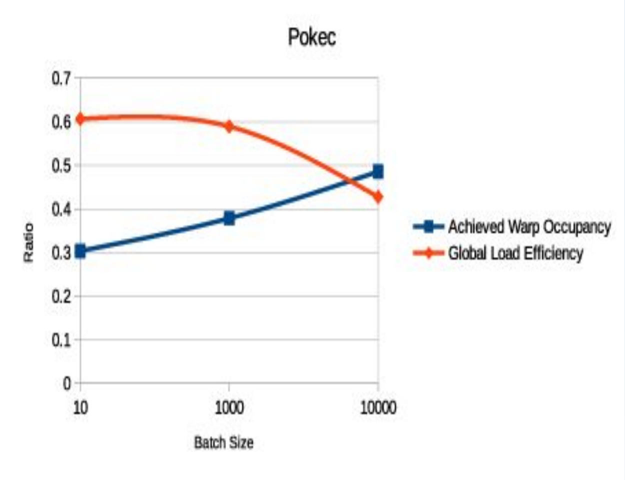
NLP
80
70
60
50
40
30
20
10
0
Time(ms)
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10
Batches
Preprocessing Time
Transfer and Kernel Time

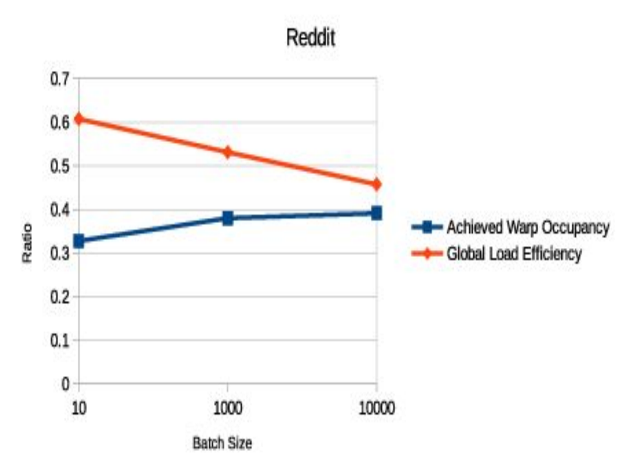
Resource Utilization

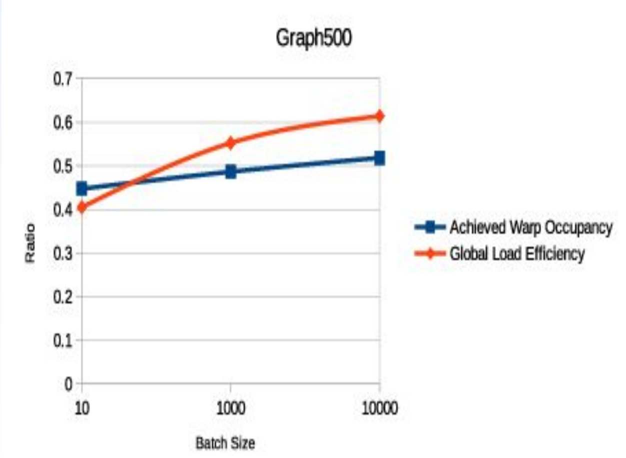
We did the profiling in order to know the consumption of the resources of the hardwares consumed, i.e the efficiency of the memory and utilization of the threads. Therefore,
we did the profiling for GPU using nvprof of CUDA toolkit. There are various profiling metrics that can be used in order to know the resource consumption of the GPU. Here in our experiment we used two profiling metrics of CUDA:

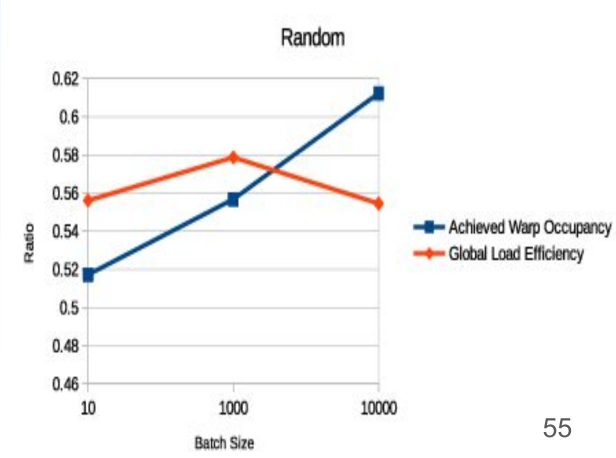
1)**Warp Occupancy** ,means
the number of warps used during the GPU kernel calls.

2) **Global load efficiency** ,means the load on the global memory or the global memory load throughput to the maximum load throughput.









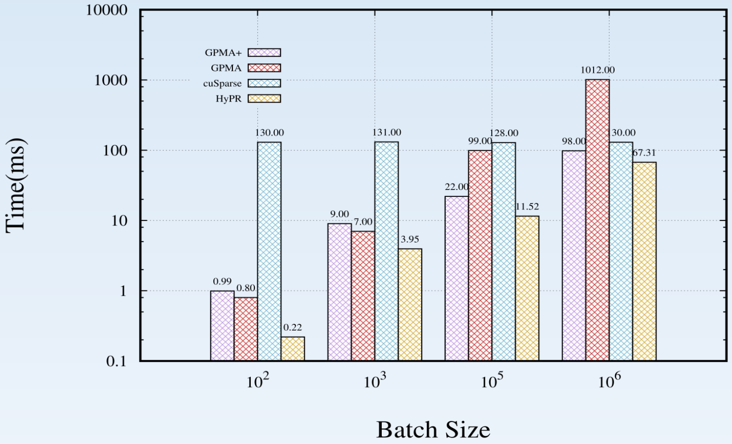
Comparative Analysis

In this subsection, we have compared our parallel approach(HyPR) with existing different updates algorithms, i.e GPMA[3], GPMA+[3],cusparse[3]. From the experiment we can find that our Hybrid approach shows better result. It is because our approach computation is divided into two parts. One part is preprocessing and the other part is computation by the GPU. Using the overlapping concept between the CPU preprocessing and GPU preprocessing we achieved the minimal time computation of PR for the evolving graph.

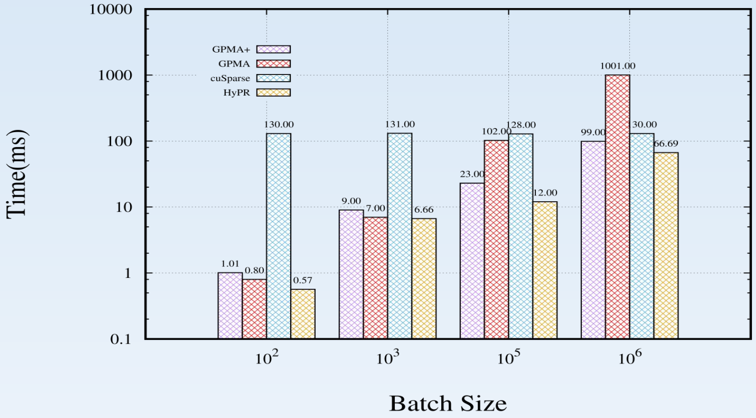
Thus, we don't need to do the intensive computation of Pagerank for the incoming batch again and again over the period of time. Instead we can preprocess the new batch with existing old batch and calculate the affected nodes and do the computation on the affected nodes accordingly. Now we have calculated the performance comparison with sliding window size of 10^2,10^3 ,10^5 ,10^6 . The comparison is made with the existing GPU based solution with our hybrid parallel approach.

Comparative Analysis

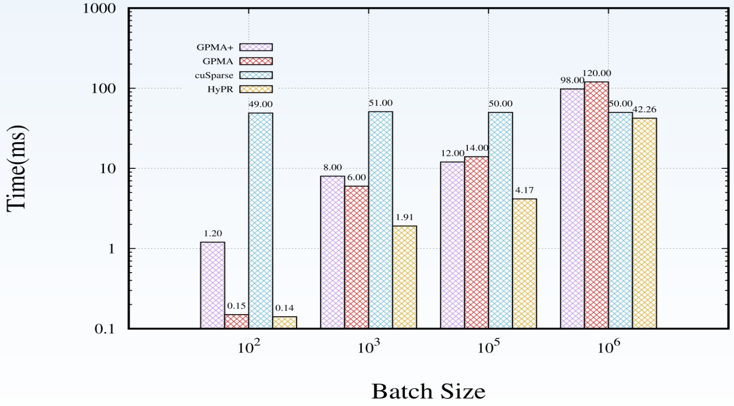
Graph500



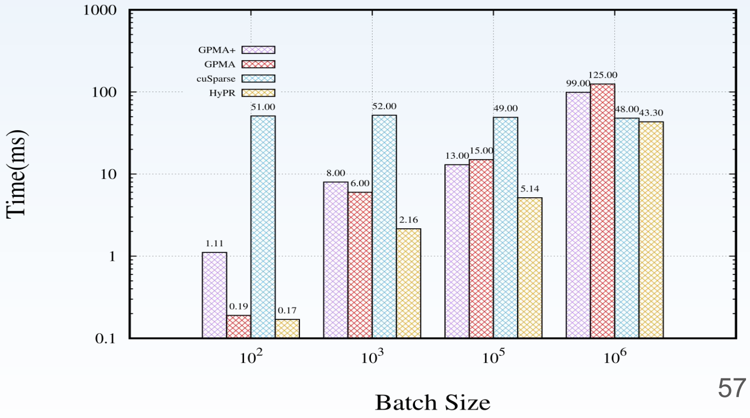
Random



Pokec



Reddit



Present Progress Work

1) Implemented the parallel Pagerank computation in incremental fashion on Evolving Graph using CPU-GPU.
2) Pre processing of the upcoming Batch when GPU is busy with computation of Pagerank and Scaling of the nodes of the previous batch.
3) In the results we show significant benefits of up to 2x over existing alternatives. Our solution also shows the scalability of our approach to networks of reasonably large sizes.

Future Work

In the near future, we plan on investigating solutions involving multiple GPUs, and multi-socket CPUs in a single platform. Additionally non-volatile storage for staging can be explored for improving I/O bottlenecks.

References

[1] Wentian Guo, Yuchen Li, Mo Sha, Kian-Lee Tan. Parallel Per-sonalized PageRank on Dynamic Graphs. PVLDB, 11(1): 9 3-106, 2017.

[2] PageRank *https://en.wikipedia.org/wiki/PageRank*. accessed 15-August-2019.

[3] Sha, M., Li, Y., He, B., and Tan, K.-L. (2017). Accelerating dynamic graph analytics on gpus. Proceedings of the VLDB Endowment, 11(1):107–120.

[4] Praveen K. , Vamshi Krishna K., Anil Sri Harsha B., S. Balasubramanian, P.K. Baruah Cost Efficient PageRank Computation using GPU

[5] PageRank *https://www.geeksforgeeks.org/page-rank-algorithm-implementation/* accessed 21-August-2019

[7] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

Thank You!!