Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master

Scalable Sub-graph Centric Distributed Influence Maximization

Anita Kaklotar

Course of Study:	Computer Science
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Heiko Geppert, M.Sc.
Commenced:	May 22, 2022
Completed:	Nov 22, 2022

Abstract

A group of socially connected persons with specific interaction or contact patterns connected through one or more online relationships can be referred to as an online social network. People can share information and communicate with their family and friends on online social networks. A lot of focus has been placed on information diffusion as a result of the popularity of online social networks, as a piece of information could spread widely through word-of-mouth among friends in the network. This diffusion phenomenon has demonstrated use in a variety of applications. The subset of influential users in the network is determined by the influence maximization (IM) problem to offer solutions for practical issues such as outbreak detection and viral marketing. Real-world graphs have become significantly more complicated and larger in recent years. Particularly, global social networks can have up to a billion connections and hundreds of millions of users. Thus, for problems like influence maximization on social networks, even algorithms with reasonable time or space complexity encounter difficulties when working with large-scale networks. In this thesis, we look at the scalability of influence maximization algorithms on large-scale networks. To address the scalability issue with influence maximization (IM) algorithms we implemented two distributed IM algorithms. The first distributed IM method we provide is based on the Update Approximation (UA) algorithm. The distributed version of UA provides an influence spread almost near to the influence spread provided with the original UA algorithm in most of the cases along with a reduction in memory usage between 13-30 % for different datasets. The other distributed algorithm we implemented is based on reverse influence sampling. We implemented a distributed version of the IMM algorithm using the existing Influence Maximization with Martingale(IMM) algorithms and parameter settings. The influence spread achieved in the case of distributed IMM is less compared to centralized IMM, but the reduction in memory usage is almost 22–25 % for different datasets.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In the study of complex networks, diffusion is a key phenomenon that helps to model the spread of ideas, adoption of a product, or spread of disease. The most studied networks are social networks, with the goal of understanding the information spread within them. The development of the Internet and blogs has made it possible to gather actual large-scale social networks. A social network in this context is a network of connections and interactions among social elements, such as people, groups, and organizations. Email networks and friendship networks are a few examples. These networks illustrate how a piece of information spreads across social networks from users to their friends. Users may efficiently converse and exchange information with their family and friends using social networks. Many global social networking websites, such as Facebook and Twitter, are increasingly popular nowadays because users may join disparate small-scale social networks and share their views and comments with their peers. In 2018, Twitter hit 335 million users overall [Staa], and Facebook surpassed 2.2 billion users in the same year [Stab]. As a result, marketing on social media sites has a lot of potentials to be far more effective than conventional marketing strategies [CPL12]. The study of influence propagation and maximization in online social networks has received a lot of attention recently due to the potential applications of this problem in a variety of fields, including personalized recommendation [STLS06], selecting influential twitters [WLJH10], target advertisement [LZT15], selecting informative blogs [LKG+07], feed ranking [IBC10], etc. Each instance shares the trait that local interactions between individuals might result in epidemic effects. This is the concept of word-of-mouth marketing, in which details about a product are spread through connections between people. A viral marketing campaign is a prominent application of this idea.

The aim is to identify a small number of targets which are the early adapters to start a propagation that results in the maximal information spread. This application raises an algorithmic issue: how can we decide which persons to target to maximize the size of a resultant cascade in a given social network? It is not always possible to use maximum targets to spread the information as there might be constraints for example, due to advertising budgets. The community overlap of the initial targets is another problem. For instance, using the two influencers with the biggest influence could result in a relatively low number of individuals being reached because they mostly have the same audience. Hence, the objective is to effectively discover a suitable collection of k nodes with which to seed a diffusion process, assuming that there is a limit k on the number of nodes to targets [KKT03].

Numerous models of influence spread have been researched for this issue, and as a result, polynomial-time algorithms with constant approximations have been created. Relevant networks for this issue can be extremely large, with billions of edges. An influence maximization algorithm's runtime is therefore a key factor. This is made worse by the fact that social network data and influence factors frequently change, forcing periodic recalculations. These factors make near-linear runtime a reasonable requirement for algorithms that operate on enormous amounts of network data.

The other major issue while solving the influence maximization (IM) problem is the scalability of the algorithm when processing large data sets. Social media graphs are getting bigger and bigger. The aim is to have scalable IM solutions that can be quickly implemented and still be scalable in the future. There have been numerous attempts to investigate scalable algorithms in large networks. To maximize large-scale online influence, we must deal with processing massive data sets that are always growing and that must be examined in a fair amount of time. Reducing the size of the data sets is one remedy while maintaining the capacity and diffusion of information [OSFK17]. Another strategy to scale the IM method is to develop a distributed version of it. To address the scalability difficulties of centralized algorithms, we construct a distributed influence maximization algorithm and attempt to balance the runtime and influence spread.

We use a distributed version in this study of the IM problem to address the scalability issue with influence maximization (IM) algorithms. The distributed version of the IM algorithm works with graph partitions created by vertex-cut partitioning. The first distributed IM method we provide is based on Update Approximation (UA) [GBR21], a very recent IM algorithm. The distributed version of UA provides an influence spread almost equal to the influence spread provided by the original UA algorithm. Then, to lessen the burden of communication overhead, an optimized version of the Distributed UA approach is suggested. In addition, we explored the Reverse Influence spread technique, a well-known technique to solve the IM problem. We implemented a distributed version of the Influence Maximization with Martingale (IMM) algorithm [TSX15] using the existing IMM algorithms and parameter settings.

Summarized, we make the following contributions:

- a scalable, novel implementation of the distributed UA algorithm built upon the existing distributed framework
- an evaluation of distributed UA against the state-of-the-art
- experimental study and implementation of the distributed version of IMM
- an evaluation of distributed IMM against the state-of-the-art

This thesis is structured as follows: Chapter 2 introduces the required background knowledge to understand the work, including details about the basics of graph theory and the details of the IM problem. It also provides an overview of the existing UA and IMM algorithms, which are used as the basis for the implementation of distributed IM algorithms. Chapter 3 discusses related work in the area of the algorithms developed to solve the influence maximization problem. In Chapter 4, we introduce the distributed version of the IM algorithms and how they work. The distributed algorithm's implementation is shown in Chapter 5. It covers the specifics of the preprocessing of the graph file needed by the algorithm, the BSP model, and current problems with distributed frameworks and how to fix them. Chapter 6 presents the evaluation and results of the work. It presents the evaluation of the distributed UA and distributed IMM algorithms against the state-of-the-art. Chapter 7 wraps up the thesis by summarizing its findings and contributions and offering potential avenues for further research.

2 Preliminaries

This chapter provides an overview of the IM problem. First, we introduce the notion of a graph and how it is used for the influence maximization problem. Next, we explain the seed set, active nodes, and diffusion models, namely Threshold Models and Cascading Models. Finally, we present the actual problem formulation and the greedy framework commonly used to solve it, along with existing UA and IMM algorithms.

2.1 Graph

There are two primary parts that make up a graph or network G: a set of vertices, denoted by V, and the edges that connect those vertices E. Hence, graph G is represented as $G(V, E)$. A directed, weighted graph is used to model the IM problem.

Definition 2.1.1
Influence graph: A directed weighted graph $G(V, E, W)$ is a graph in which the edges have a direction with edge weights. V represents the set of nodes of the graph and E the edges where $E = \{(u, v) \in V \times V \mid u \neq v\}$. The function $W : E \rightarrow$ maps the edges to a numerical value. In case of IM this weight function that assigns each edge e = (u, v) value in (0,1) and represents the influence probability.

A social network could be represented using this weighted directed graph $G(V, E, W)$, which shows users and their relationships in the network with specific tie strength [SSVS22]. The graph also represents how each node influences one another in terms of influence strength.

Definition 2.1.2
Neighbours: Neighbours of a node u, N(u) are defined as the set of nodes v such that $N(u) = \{v \mid (u, v) \in E \vee (v,u) \in E \}$. In-neighbours of u can be represented as $N_{in}(u)$ which are the nodes with an edge pointing to u. Where $N_{in}(u) = \{v \mid (v, u) \in E\}$. The number of in-neighbours of a node defines the in-degree of a node. The outgoing edges of the node represent out-neighbours of u and can be represented as $N_{out}(u)$. Where $N_{out}(u) = \{v \mid (u, v) \in E\}$. The out-degree of a node is defined as the number of out-neighbours of a node.

An edge (v, u) states that node v has some influence on node u. As the graph is a directed graph, the edge set is the collection of ordered pairs of nodes, i.e., $E(v, u) \neq E(u, v)$. The edge may or may not exist in both directions.

2.2 IM Problem

IM problem aims to obtain the initial k seed users which when influenced provide maximum influence spread under a certain diffusion model. This section provides information on different diffusion models, IM problem statement, and the commonly used greedy framework.

2.2.1 Diffusion Models

A diffusion model is a method to simulate the influence spread in the network. A set of nodes initially used for the propagation of information spreading are called seed nodes, represented by S where $S \subseteq V$. At a given timestamp t every node $u \in V$ in a graph is associated with either of the two states: active or inactive. An active node u can be from the seed set S or another node v from the node set V that has been activated by an already active node. An inactive node is one that is not yet influenced by its neighbors. Initially, all the nodes are inactive except for the seed nodes. Seed nodes start the diffusion process at timestamp t = 0 and start influencing their neighbors. These activated nodes try to influence their neighbors, and the process continues until no new nodes can be activated further. Different diffusion models have different mechanisms for capturing the active state from the inactive state. The number of active nodes generated after the implementation of the diffusion process defines the influence spread. The two most common diffusion models are Threshold models and Cascading models.

Threshold Models (TM)
In case of TM node status change from inactive to active based on some threshold value. The most commonly used threshold model is the linear threshold model (LTM). The linear threshold model is based on the idea that individuals need the consideration of multiple independent friends to change their minds. The approach is appropriate for outlining how a novel, unproven product is adopted. Once the cumulative of all of their friends' recommendations reaches a certain level, people are more likely to utilize the new product themselves [CLC13]. This is typically true when an industry adopts a new standard. In this model, every node u contains an activation threshold value $T_{\text{u}} \in (0, 1]$. The concept behind this model is that a node u becomes active iff $\sum_{v \in N_{\text{in}}(u)} W_{\text{v,u}} \geq T_{\text{u}}$ where $N_{\text{in}}(u)$ is the set of all the in-neighbors of u, i.e. the neighbors influence must be greater than the given threshold value T_{u}. The value of the user's threshold follows uniform distribution over [0,1]. Apart from this, other threshold models are also available that differ by their threshold values, like majority TM and small TM. For majority TM the threshold value is defined as $T_{\text{u}} = \frac{1}{2} D_{\text{u}}$ where D_{u} is the degree centrality of node u and for small TM, the threshold value is a small constant [SSVS22].

Cascading Models (CM)
Cascade models are used for diffusion processes, which are inspired by probability theory. The Independent cascade is most commonly employed in viral marketing and is well-suited to it. One person is sufficient to activate another person, which is the primary property of the independent cascade model (IC), which states that the influence spread is independent in each arc of the entire graph. This model can be used to simulate the spread of concepts, knowledge, infections, and other things [CLC13]. If a node u becomes active at time t, it has only one chance of activating its nonactive out neighbors $N_{\text{out}}(u)$ at timestamp t +1. The probability of activating the node depends on the edge weight. If node v becomes active, it will never become inactive again. The process will stop when no new nodes are activated. The weighted cascade model is a specialized instance of the

Algorithm 2.1 Greedy Algorithm

Input Graph:G, seeds: k
Output SeedSet: S

```
$S \leftarrow \emptyset$
while $|S| < k$ do
    max $\leftarrow$argmax$_{v \in V \setminus S \sigma(S \cup \{v\}) - \sigma(S)}$
    $S \leftarrow S \cup max$
end while
return $S$
```

independent cascade (IC) model. This model uses inverse in-degree of nodes as edge weight. Each edge from node u to v is assigned probability $\frac{1}{|IN(v)|}$ of activating v where $|IN(v)|$ denotes the in-degree of node v. In our algorithm, we use a weighted cascade model for the diffusion process.

2.2.2 Problem Statement

IM problem is to find a set S^* of k seed nodes which maximize influence spread $\sigma(S^*)$ in the given graph G, where $\sigma()$ presents the expected influence spread function. The influence spread in G can be maximized by using an information diffusion model in our case weighted cascade model that considers a positive integer k and a subset of influential users $S \subseteq V$, |S| = k such that

$$\sigma(S^*) = argmax_{S \subseteq V \wedge |S|=k} \sigma(S)$$

2.2.3 Greedy Framework

The greedy framework proposed by Kempe et al. [KKT03] is the most common framework used for influence maximization strategies. The algorithm begins by taking an empty set as a seed set. It iteratively identifies the node with the maximum marginal gain and adds this node to the seed set. In every iteration, the marginal gain is recalculated because recently selected nodes can change the value. The algorithm terminates when k distinct seed nodes are found.

Finding the optimal solution for the IM problem is not feasible because the IM problem is an NP-hard problem. Hence, to find an approximate solution, the expected influence spread function should be submodular. A function is submodular if it satisfies both the properties of monotone increasing and diminishing returns.

Property 1 Monotone Increasing: An objective function say f is said to be monotone increasing if adding element to the set can not reduce tha value of f, i.e., $f(S \cup \{u\}) \geq f(S)$ [SSVS22]
Property 2 Diminishing Return: An objective function say, f is said to be diminishing return if the marginal gain from adding an element to a set S is at least as high as the marginal gain from adding the same element to a superset of S, i.e., $f(S \cup \{u\}) - f(S) \geq f(T \cup \{u\}) - f(T) \forall u \in T$ and $S \subset T$ [SSVS22]

Kempe et al. suggest an estimating approach based on Monte Carlo simulations that can approximate the value of marginal gains with decent precision in an acceptable time to get around these high processing overheads [KKT03]. We flip a coin and eliminate e with a probability of 1 - p(e) for each edge e in G. After doing this for all edges, a resulting graph g is obtained. A node v in g is defined as reachable from the set S if a directed path leading from a node in S and terminating at v exists in g. Let R(S) represent the collection of nodes in g that may be reached from S. Kempe et al.'s study demonstrate that the expected value of influence spread of set S equals the expected size of R(S). Consequently, we can opt to estimate E[R(S)], which is significantly simpler, rather than working directly on the expectation of influence spread of set S. We first create a sufficiently big set of g samples, after which we compute R(S) for each sample. We can consider the arithmetic mean of all sample measurements to represent an estimation of the expectation of influence spread of S. Greedy framework guaranteed (1-1/e- ϵ) approximation ratio.

2.2.4 Overview of UA Algorithm

The Update Approximation algorithm is an iterative algorithm that selects seed nodes based on the influence scores in a greedy manner and "corrects"the scores up to two hops away from the seed. The next node is chosen after recalculating the score for that node, thus updating the third hop. The concept behind calculating a node scoring function Δh is that the number of weighted simple pathways of length h beginning at a node corresponds to the influence of that node on the nodes around it. The Δh score can be calculated iteratively using the neighbors' Δh-1 scores. UA algorithm calculates the scoring function using little memory. For each node within an h-hop radius of the newly chosen seed, all h recursion steps must be recalculated in order to update nodes. This is not practical since it takes a lot of pricey set intersections to collect every h-hop neighbor for a greater h. For bigger graphs or larger k, recalculating all nodes is not scalable. UA can execute accurate updates for $h \leq 3$ and approximation updates for $h > 3$.

$$\Delta_{\text{h}}(v) = \Delta_{\text{h-1}}(v) + \sum_{n \in N_{\text{out}}(v)} (w(v,n) * \Delta_{\text{h-1}}(n))$$

Updates are carried out by UA in two directions: two hops forward and one hop backward. First, within two hops of a freshly chosen seed node, Δh-1 is updated. The latter uses the CELF approach to update Δh of promising nodes. When a new seed node s is added to S, the forward update is executed. UA algorithm begins by updating the weighted degree of each of s's one hop neighbors. The Δh-1 score of the one and two hop neighbors is then updated. Therefore, the edge probability of the path from s via u (neighbors of s) to the two hop neighbor is then subtracted from the score as the algorithm iterates the neighborhoods from all neighbors u of s. The update of Δh-1 is accurate if the recursion depth is 3 or less than 3, as it represents $\Delta 2$. When h is 1, update of Δh-1 is skipped since it is not required. When h is 2, since the backward update can cover the second hop more quickly, the update of Δh-1 is skipped. The backward update simply recalculates Δh to update the Δh score of a promising node. As a result, the most recent Δh-1 scores for v and its neighbors are used. If h is more than 3, just the first three hops are updated, which introduces a tiny mistake. The authors accept this inaccuracy because it is manageable—the majority of influence propagation occurs within the first hops, anyway [STLS06] and because it keeps computation time to a minimum [GBR21].

2.2.5 Reverse Influence Sampling (RIS) Framework

Reverse-reachable (RR) sets are the basic idea behind the RIS methodology. The concept behind RIS is to take a series of random reverse samples and find probable seed nodes that appear in a larger number of samples [BBCL14]. The reasoning is that a good seed node will be part of many random reverse samples as it is reachable by many other nodes in forward propagation. With a network G = (V, E) and a diffusion model in place, an RR set $R \subseteq V$ is sampled by first randomly choosing a node $v \in V$, then replicating the diffusion process in the reverse direction and adding any nodes that were reached by the reverse simulation into R. i.e. finding nodes that could possibly influence v. Consider the following definitions:

Definition 2.2.1
Reverse Reachable Set: Let v denote a node in G and g denote a graph obtained by removing each edge (u,v) in G with a probability of 1 - $w_{u,v}$. Then, the reverse reachable set for v in g, denoted by RRg(v), is given by: RRg(v) = { u | ∃ a directed path from u to v in g }

Definition 2.2.2
Random Reverse Reachable Set: A random reverse reachable (RRR) set for v, denoted by Rv, is an RRg(v) where g is a randomly sampled graph, drawn from a distribution of graphs induced by the randomness of edge removals.

In other words, node u has the potential to affect v if node u is present in the reverse reachable set of v called Rv. Consequently, it is anticipated that a given node u will have a larger influence on the given graph if it appears in many random reverse reachable sets.

An algorithm based on RIS creates many reverse reachable sets. It uses these sets to find K seeds that can cover as many reverse-reachable sets as possible. If at least one seed is present in a RR set, we consider the RR set covered. The algorithm uses a greedy coverage approach. Hence, it iteratively chooses a seed that covers the maximum number of remaining RR sets. The core of RR set based algorithms is that, for a seed set S, the likelihood that a vertex selected randomly from graph is infected by S is the same as the probability that S covers a randomly sampled RR set. The fraction of the RR sets covered by S is a good approximation when enough reverse reachable sets are sampled.

3 Related Work

Influence maximization (IM), one of the computer science topics that is most actively researched, has seen a variety of successful strategies over the past years [DR01; KKT03; SSVS22]. We review the current methods based on their method of seed selection. We discuss a group of algorithms that relies on explicit MC simulations of the information diffusion process to determine each node's influence and choose the seed set. The other category algorithms speed up the seed selection process by relying on sampling-based exploration rather than directly evaluating each node's contribution. The last category of algorithms is based on the use of some scoring function approximation to estimate the node influence.

The influence maximization problem was first proposed by Domingos et al. [DR01]. The paper showed the use of a Markov random field for modeling the influence of users and heuristics to identify users having a large influence. The second important study on the influence maximization problem was presented by Kempe et al. [KKT03]. The study showed that the influence maximization problem is NP-hard. The paper also explains the greedy algorithm, which provides a (1-1/e- ϵ)-approximation for both LT and IC models. Furthermore, they explained through experiments that their greedy algorithm outperforms the classic degree and centrality-based heuristics in influence spread. Kempe et al.'s algorithm is simple and efficient, but it has a drawback of a high time complexity, which makes it difficult for large-scale graphs.

Leskovec et al. [LKG+07] present the Cost-Effective Lazy Forward (CELF) strategy, which performs the optimization. And drastically reduces the number of evaluations on the influence spread of vertices by utilizing the objective function submodularity property. However, for the larger graphs, the algorithm still takes a few hours to complete, so it is still not a sufficiently scalable algorithm for large-scale networks. The paper by Goyal et al. [GLL11b] presents improved the version of the CELF algorithm, again by using the submodular property of the objective function and gives a new algorithm CELF++.

W. Chen et al. [CWY09] proposed an algorithm named NewGreedyIC which is based on a forward influence sketch under the IC model. In forward influence sketch, the idea is to generate a sketch by extracting the subgraph wrt. to a specific diffusion model. These subgraphs are then used to accurately find the influence spread of seed set S. A sketch can be generated under the IC model by eliminating edge E(u,v) of a graph G(V, E) with the probability 1-$p_{u,v}$. This results in a subgraph of G called G_i. Depending on a θ value sketches are constructed as $\{G_1, G_2, G_3, .., G_\theta\}$. From this $\sigma(S)$ is calculated as an average number of users reached by S on these sketches where S is the seed set. NewGreedyIC generates several sketches and simultaneously evaluates $\sigma(S \cup \{u\})$ - $\sigma(S)$ for all $u \in V$ not in S. NewGreedyIC provides a boost in the performance.

Tang et al. [TXS14] proposed TIM and TIM+, which are inspired by Borgs et al. Both approaches produce (1-1/e- ϵ)-approximation. Their results provide better results in response to handling large graphs. The triggering model, a generic diffusion model that includes both IC and LT as special examples, is supported by TIM. To significantly increase its empirical efficiency while maintaining its asymptotic performance, TIM incorporates novel heuristics into its practice. There are two issues with TIM and TIM+. The first is that it's estimated lower bound of OPT, where OPT is the ideal maximum effect of k nodes, is quite conservative, resulting in a large number of useless samples; the second is that computing the lower bound takes a long time.

Sketch-based Influence Maximization (SKIM) proposed by Cohen et al. [CDPW14] is an another approach that works on constructed sketches using bottom-K minHash to speed up the influence estimation. Bottom-K min hash constructs the sketches, which consist of k minimum ranked items with the use of some hash function. The algorithm uses reverse breadth-first traversal (BFS) walks on the generated sketches and for the number of candidates, and seed sets it updates the bottom-K minHash values at the same time. SKIM utilizes multi-CPU parallelization.

In [KS06], Kimura and Saito proposed two natural models for information diffusion in a social network, called the Shortest-Path Model (SPM) and SP1 Model (SP1M). The author assumes that the influence spread process is influenced by only the first two shortest paths. For computing the spread of influence the recursive computation of the Dijkstra shortest-path method is utilized. In SP1M an objective function is estimated by approximation strategy and it does not use MC simulations. The authors showed the proposed models can be scalable, faster than the ICM. But it has poor scalability and becomes infeasible for graphs in larger sizes. The algorithm can take a large amount of time to select 50 seeds when the graph size reaches a few hundred thousand.

Chen et al. [CWW10] proposed an algorithm inspired by SP1M named maximum influence arborescence (MIA). To estimate influence spread it maintains the local arborescence structure and restricts the influence diffusion of a node to the local tree structure. Influence estimation becomes much faster as the influence of a node in a tree can be calculated efficiently. Also, the algorithm computes influence spread by considering only the highest propagation probability paths. The authors also present another variant of MIA, named prefix excluding MIA (PMIA), which is computationally more efficient.

Another algorithm having a similar basic idea as PMIA is LDAG [CYZ10] but it uses the LT model. LDAG restricts the graph to be DAG because in the LT model the influence can be calculated efficiently in directed acyclic graphs. It uses Dijkstra's shortest path algorithm to construct Local DAG(v,θ) which contains nodes having influence with probability θ on the node v. Using this, LDAG influence spread of any node can be computed efficiently. In the case of large graph LDAG process is memory and computationally intensive.

SimPath is another algorithm that is based on the LT model, proposed by Goyal et al. [GLL11a]. SimPath is based on the concept that under the LT model influence spread can be calculated by enumerating all simple paths starting from each node in the set. SIMPATH prunes the paths with probability less than θ to restrict the enumeration to a small neighborhood. The influence of a node u is calculated by summing up these simple paths. In order to calculate influence spread $\sigma(S)$ is calculated by adding the influence of every node $u \in S$ in the subgraph induced by V$\backslash S \cup \{$u$\}$

Kim proposed an influence spread approximation algorithm based on the IC model named Independent path algorithm (IPA) [YKK13]. The main concept is influence paths, which simplifies influence spread approximation in IPA and also makes algorithm parallelization easy. By combining IPA and the CELF greedy algorithm a seed node set of influence maximization problems was obtained.

Liu et al. [LXC+14] proposed a heuristic-based method called Group-PageRank that can determine the influence spread in a linear amount of time. The algorithm sums up the PageRank scores of all nodes in a node set S and removes the mutual influence between the nodes.

Jung et al. [JHC12] proposed a novel algorithm named IRIE that combines the advantages of influence ranking (IR) and influence estimation (IE) methods. This algorithm determines the influence spread recursively depending on the influence spread of each node's neighbors. For each node u in the graph, IRIE develops a set of n linear equations with n variables to calculate the effect r(u) of that node. A node's influence r(u) includes both its impact on itself, which is 1 in this case, and the total of the influences it propagates to all of its neighbors. These equations can be solved faster by an iterative method. The node with the highest influence is added to the seed set S and the formula for r(u) is updated. After updating and resolving the system of linear equations k times, IRIE returns the seed set S. Compared to the General Greedy algorithm, the IRIE approach has relatively little memory overhead, but it has a smaller influence spread.

EASYIM [GAR16] algorithm is based on enumerating simple paths for both IC and LT models. It counts the influence path within length l to calculate the influence of each node and also checks the overlaps between various paths for better accuracy. It achieves better performance by using an iterative method assembling IRIE for global influence estimation.

Reverse reachable (RR) sets, a new idea was first introduced by Borgs et al.[BBCL14]. They assumed that it is not necessary to operate on the whole graph for the estimation of influence spread. Another method that takes advantage of these RRR sets named Reverse Influence Sampling(RIS) was also introduced by Borgs et al. The approach first creates several random reverse accessible sets and then computes a set of k seed vertices that covers the greatest number of those samples. The quantity of samples that must be generated in this case, is determined by a user-defined threshold that is based on the total number of vertices and edges that must be traversed. The algorithm guarantee approximation of (1-1/e- ϵ). The barrier here is to find the correct threshold value that provides the correctness of the algorithm as well as helps in not generating too many RR sets.

The RIS method is the foundation of an influence maximization strategy named IMM [TSX15]. IMM, at its most basic level, comprises the following two phases: 1) Sampling: Random RR sets are generated in this phase and are inserted into set R until the stopping condition is satisfied 2) Node Selection: This step creates a size-k node set S that covers a large number of RR sets in R using the conventional greedy technique for maximum coverage. The final outcome is then returned as a set of k seed nodes.

Geppert et al. [GBR21] proposed a new algorithm named Update Approximation(UA) for iterative influence maximization, which is capable of influencing a large number of nodes within a few seconds on a billion-scale graph. UA algorithm selects seed nodes based on Δ scores. After this, it corrects the scores within a 2 hop radius around the seed node. The score is recalculated when selecting the next node which helps in performing a cheap third hop update. The average time complexity of UA is significantly reduced compared to iteration-based algorithms.

For the first time Zhou. Zong et al. [ZLH14] proposed distributed algorithm called distributed influence maximization algorithm. Authors proposed three versions of dIRIE called dIRIEb, dIRIEi, dIRIE. dIRIEb is a distributed basic version of centralized IRIE. The incremental approach is used in the version of dIRIEi. In the last version, dIRIEr authors used the reservoir strategy to induce stability by reducing communication traffic. Large-scale network experiments reveal that dIRIEr outperforms state-of-the-art centralized IRIE in terms of speedup and can handle enormous graphs where centralized IRIE is impractical.

The more the sampling better the influence spreads. It makes sense that more samples would aid in a better understanding of information dissemination and facilitate the selection of the most influential users. On the other hand, processing huge volumes of sample data are very difficult. Estimating the impact spread for large-scale graphs may require a significant amount of processing resources, such as CPU and memory, and may not be possible on a single conventional system. It makes sense to use distributed algorithms to split the burden across numerous machines and speed up processing while dealing with enormous amounts of data. Distributed IM will help scaling of the IM algorithm, which is a major issue in most of the existing algorithms when used for larger graphs.

4 Distributed IM Algorithms

In this chapter, we first introduce the distributed version of the UA algorithm. Then, we introduce an alternative distributed UA approach that has a shorter runtime but a narrower influence spread than the first version. In the end, we explain the distributed version of the IMM algorithm.

4.1 Distributed UA

Distributed UA works with partitions of a graph. The idea behind the distributed UA algorithm is to find the maximum influencing vertex at each step from all the partitions. The influence values are calculated using a node scoring function. The first step is to calculate these node score values. The next step is to perform forward and backward updates of the UA algorithm [GBR21] independently on all the computing nodes. The third step is to find the seed vertex with the highest score. The following section explains these three steps in further detail.

Every computing node starts with a given partition number. A computing node that is given partition number zero is considered to be the master server. It handles all the synchronization steps required for the superstep cycle of the Bulk Synchronous Parallel (BSP) model [Val11]. It is also responsible for selecting the seed vertices and aggregating information received from other computing nodes. V_{p} represents all the vertices in partition P. A vertex may have several duplicates in other machines, but edges are not duplicated (as the graph is vertex cut). We pick one vertex copy to be the master copy and the others to be the mirrors. The vertex's master copy's partition is the vertex main partition number. ${V_{\mathrm{p}}}^{\mathrm{mi}}$ represents the mirror vertices in a partition P that are present on the current computing node, but the main partition number is not the same as the computing node partition number. ${V_{\mathrm{p}}}^{\mathrm{ma}}$ shows the master vertices in a partition P that are present on the current computing node, for which the main partition number is the same as the computing node partition number. Consider the following example: we have an edge from vertex A to vertex B and the edge is placed on partition 0. For vertex A, the algorithm calculates the main partition number as 1. Then the computing node with partition number 1 is responsible for aggregating the updates from vertex A's other replicas and sending them the updated value. So the computing node with partition number 1 will have vertex A in ${V_1}^{\mathrm{ma}}$ and the computing node with partition 0 will have vertex A in ${V_0}^{\mathrm{mi}}$.

Every partition calculates the main partition for every vertex having an edge on that partition and also stores the information about edges and other replicas of these vertices. For other vertices, they just store the partition id. Storing the partition id of a vertex is still scalable compared to storing all the edges of a vertex since the number of edges introduces scaling issues for large graphs.

We assume that every partition can communicate with every other partition. That serves as the algorithm's important precondition. At each computing node, for all the vertices in V_{p} we calculate the vertex's initial weight by combing weights of outgoing edges from the vertex. For the vertices present in ${V_{\mathrm{p}}}^{\mathrm{mi}}$ we send the partial result to the vertex's main partition computing node. When the main partition node receives this value, it aggregates the weight of that vertex. Every computing node synchronizes these initial aggregated weights of the vertices present in $V_\mathrm{p}{}^\mathrm{ma}$ to all the other replicas of the vertices. Lines 2-14 in algorithm 4.1 represent these steps.

Each computing node uses these aggregated weights to calculate the scoring function Δ_h of the vertices present in $V_\mathrm{p}{}^\mathrm{ma}$ and sends the partial results for the vertices present in $V_\mathrm{p}{}^\mathrm{mi}$ to the main partition of the vertex. The idea is that a vertex influences its surrounding vertices by the number of weighted simple paths of length h starting at the vertex. The weighted score of each vertex's neighbors is continuously added to show the influence each vertex has within a certain hop distance [GBR21].

$$\Delta_\mathrm{h}(v) = \Delta_\mathrm{h\text{-}1}(v) + \sum_{n \in N_\mathrm{out}(v)} (w(v,n) * \Delta_\mathrm{h\text{-}1}(n))$$

After every iteration, all the computing nodes send the delta score of vertices present in $V_\mathrm{p}{}^\mathrm{ma}$ to all the other replicas of the vertices, so that every computing node has the same delta scores for all the vertices present in V_p. This part is represented as *performWeightedDegreeIteration* function in algorithm 4.2. The runtime of an algorithm is significantly affected by the number of times we synchronize the data. To achieve larger influence spreads, we synchronize data at each step of the algorithm.

After the calculation of delta scores, the final step is to select the k seed nodes which is stated in algorithm 4.3. The master server chooses the vertex with the highest score and sends this vertex's details to all the other slaves. Lines 4-6 in algorithm 4.3 show this step. All the computing nodes perform a forward update on the received vertex. The next step is to perform a backward update on the highest Δ_h value vertex available on each computing node. This part of the slave server is represented by lines 8-10 in algorithm 4.3. The forward and backward update is the same as in the Update approximation algorithm [GBR21]. The next step is to find the highest Δ_h vertex at every computing node and send it to the master server. Again, the master server finds the maximum of these received values and sends this vertex's details to all the other slaves. The process repeats until the k seeds are found.

4.2 Optimized Distributed UA

The synchronization of vertices values present in $V_\mathrm{p}{}^\mathrm{ma}$ to all the other replicas of the vertices greatly influences the runtime of the Distributed UA algorithm. For example, consider that we have four partitions. At partition zero, if we have 4000 vertices with the main partition as zero, then at every synchronization step the zeroth partition will send 4000 * (number of replicas of a vertex) messages. Similarly, other partitions will also send messages to all the other replicas of vertices. So, in total, n * $\mathrm{V_p}{}^\mathrm{ma}$ * (number of replicas of vertex) messages are sent at each synchronization step. Assuming every partition has roughly the same number of vertices in $\mathrm{V_p}{}^\mathrm{ma}$.

In the previous distributed algorithm version, we performed one synchronization of the initial weight calculation of vertices at line 12 in Algorithm 4.1, and after each iteration, we performed synchronization (line 13 in Algorithm 4.2) of delta scores in order to get the correct calculation for

Algorithm 4.1 Distributed UA Algorithm

Input Graph:G with weighted edges and partition number, seeds: k
Output SeedSet: S

```
1: $S \leftarrow \emptyset$
2: $\forall v \in V_p$ calculate initial vertex weight (summing outgoing edge weight of vertex)
3: if $v \in {V_p}^{mi}$ then
4:     send partial result vertex weight to master partition
5: end if
6: perform synchronization barrier step
7: for $v \in V_p \setminus {V_p}^{mi}$ do
8:     aggregate all received values of a vertex
9: end for
10: perform synchronization barrier step.
11: for $v \in {V_p}^{ma}$ do
12:     send the vertex weight to all the other replica of the vertices
13: end for
14: perform synchronization barrier step.
15: while given number of iteration do
16:     performWeightedDegreeIteration()
17: end while
18: while $|S| < k$ do
19:     nodeSelection()
20: end while
21: return $S$
```

Algorithm 4.2 performWeightedDegreeIteration()

```
1: for $v \in V_p$ do
2:     calculate $\Delta_h(v)$
3:     if $v \in {V_p}^{mi}$ then
4:         send partial result $\Delta_h(v)$ to master partition
5:     end if
6: end for
7: perform synchronization barrier step
8: for $v \in V_p \setminus {V_p}^{mi}$ do
9:     update $\Delta_h(v)$ with all received values
10: end for
11: perform synchronization barrier step
12: for $v \in {V_p}^{ma}$ do
13:     send the final calculation $\Delta_h(v)$ to all the other replica of the vertices
14: end for
```

Algorithm 4.3 nodeSelection()

```
1: find $argmax_{v \in V \setminus S} \Delta_h(v)$ at each server
2: send the vertex and weight to Master server.
3: perform synchronization barrier step
4: At Master server do
5: max $\leftarrow$argmax$_{v \in V \setminus S}$ $\Delta_h(v)$ from all the received values
6: $S \leftarrow S \cup max$
7: sendForwardUpate(max)
8: At Slave server do
9: forwardUpdate(received vertex)
10: perform backwardUpdate(next heighest $\Delta_h(v)$)
```

the influence spread. But this results in increasing runtime as the sending of data to other computing nodes is time-consuming. In order to reduce this overhead, we propose an optimized version of the distributed algorithm. The algorithm trades runtime against the influence spread.

The optimized distributed UA version sends the partial results of vertices in ${V_p}^{mi}$ the same number of times as the distributed UA algorithm. But the synchronization of vertices values in ${V_p}^{ma}$ is done only after the final calculation of delta scores. Here we skip the synchronization steps at line no. 12 in algorithm 4.1 and line no. 13 in algorithm 4.2. I.e., we calculate delta scores of the vertices in ${V_p}^{ma}$ locally at every computing node and then synchronize them to all the other replicas of the vertices. So here we have only one synchronization step that is performed at the end of the *performWeightedDegreeIteration* function.

4.3 Distributed IMM Algorithm

The idea behind the distributed IMM algorithm is to independently work on the vertices that are part of the partition instead of taking entire graph vertices. Within a single partition, the distributed IMM algorithm acts similarly to the original IMM influence maximization strategy. The parameter settings for the algorithm are taken from the existing IMM algorithm [TSX15].

In the distributed IMM algorithm, we run multiple instances of the IMM algorithm on different machines. Each instance first creates RR sets similar to the IMM algorithm. But it selects random vertices which are part of the partition V_p and considers only the edges present on P for the RR sketches. The next step is to perform seed vertices selection. At each instance, we select K seed vertices using the generated RR sets. This part is done in *RunIMM* (line 3 in Algorithm 4.4) at each partition, which results in k seed vertices per instance. Every instance sends its best k vertices along with the counter value to the master server. The counter value denotes the value that shows in how many RR sets the vertex appears. The master server selects the k vertices with the highest counter value from all the received vertices and returns them as the final seed set. These steps at the master server are shown in lines 9–10 of Algorithm 4.4.

Algorithm 4.4 Distributed IMM Algorithm

Input Graph:G with weighted edges and partition number: P, seeds: k
Output SeedSet: S

```
1:  $S \leftarrow \emptyset$
2:  At each server $\forall v \in V_{\mathrm{p}}$
3:  k = RUNIMM()
4:  At slave server do
5:  Indenitfy k nodes with highest counter value
6:  sendToMaster(k nodes, counter values)
7:  perform synchronization barrier step
8:  At master server
9:  find the k nodes with maximum counter value received from all the server.
10: $S \leftarrow knodes$
11: return $S$
```

As the distributed version works with only vertices present in its partition, the number of vertices in a RR set is lower compared to the centralized IMM. This is because when we add a random vertex to a RR set and find its inverse edge, the inverse edge exists, but it might happen that this is not present on the given partition, hence the destination vertex of the inverse edge of the random vertex is not added to the RR set. Hence, the size of the RR set is less compared to the centralized IMM.

To increase the number of vertices inside a RR set, we created a modified version of the distributed IMM. In this version of IMM, we selected two random vertices instead of one when generating the RR set. The idea here is to increase the FR value. The FR value is the fraction of the RR sets covered by the vertices.

$$FR = \frac{Number\ of\ RR\ sets\ covered}{Total\ number\ of\ RR\ sets\ generated}$$

As we select two random vertices, more vertices will be present in RR sets. Hence, we will be able to cover more RR sets as compared to the previous version of distributed IMM, and we will get a higher value of FR at an early stage. This FR value is used in the stopping criteria for generating RR sets. The higher the FR value the earlier is the termination of generating RR sets.

The idea of the IMM algorithm is that when the number of RR sets θ in R is sufficiently large, a node selection of K seed vertices ensures a (1-1/e- ϵ)- approximation to the influence maximization problem with a high probability [TSX15]. We have used the same calculation as stated in the original IMM algorithm for calculating θ value.

The centralized IMM algorithm works in iterations. In the i-th iteration, the algorithm computes $x = \frac{n}{2^{\mathrm{i}}}$, where n is the number of vertices. After this it calculates $\theta_{\mathrm{i}} = \frac{\lambda'}{x}$ In order to reduce the total number of RR sets generated such that the influence spread is not affected, we have evaluated the distributed IMM with different values of θ and we found that θ when reduced to 2.5 of its original value gives the same influence spread as that of the original distributed IMM version i.e $\theta_{\mathrm{i}} = \frac{\theta_{\mathrm{i}}}{2.5}$. This value of θ was established during early evaluations. There may be more accurate values to provide better outcomes. The idea here was to reduce the total number of RR sets generated without affecting the influence spread so that it reduces the runtime of the algorithm.

5 Implementation Overview

The section gives details about the implementation of distributed Update Approximation algorithm. In this section, we first discuss the format of the input graph file required for the distributed UA algorithm. Then we give an overview of the bulk synchronous model, which is the basic technique that is used in a distributed framework. We also explain the issues with the existing distributed framework and how we fix them.

5.1 Input Graph file structure

The first step is to get the input graph file in the required format, which is an edge list with edge weights and partitions. The graph file downloaded from the Stanford Large Network Dataset Collection (SNAP) is a graph file with an edge list [LK14]. This file is then given as input to the graph partition algorithm as well as the edge weight calculation algorithm. The edge weight calculation is a python script that uses the inverse in-degree method to calculate the edge weight for all the edges present in the given graph file. This script outputs a graph file having the source, destination, and corresponding edge weight. To partition the graph, the publicly available GraphPartitioner algorithm [1] is used, which requires an edge list graph file as input. The GraphPartitioner provides NE and HDRF partitioning algorithms to perform the partitioning. The output of the script is a file stating the partition number for all the edges in a given graph file. Petroni et al. [PQD+15] proposed a novel stream-based graph partitioning technique named High-Degree (are) Replicated First (HDRF) for distributed graph-computing frameworks. In this research, the foundation is a greedy vertex-cut method that makes use of vertex degree data. Chenzi Zhang et al. [ZWL+17] proposed a heuristic called NE (Neighbor Expansion) for partitioning large graphs. This model greedily maximizes edge locality.

The output files from the edge weight calculation algorithm and the partition algorithm are combined into a single file that represents an edge list with edge weight and its partition. This is the required form of a graph file as input to the distributed UA algorithm.

5.2 Bulk Synchronous Parallel Model

The distributed framework works on the Bulk Synchronous Parallel model [Val11]. Bulk Synchronous Parallel (BSP) is a parallel computing architecture and programming model. Supersteps are used to segment computation into several phases. Each superstep consists of a collection of concurrently executing processes that share the same code and produce messages that are delivered

[1] https://github.com/zongshenmu/GraphPartitioners

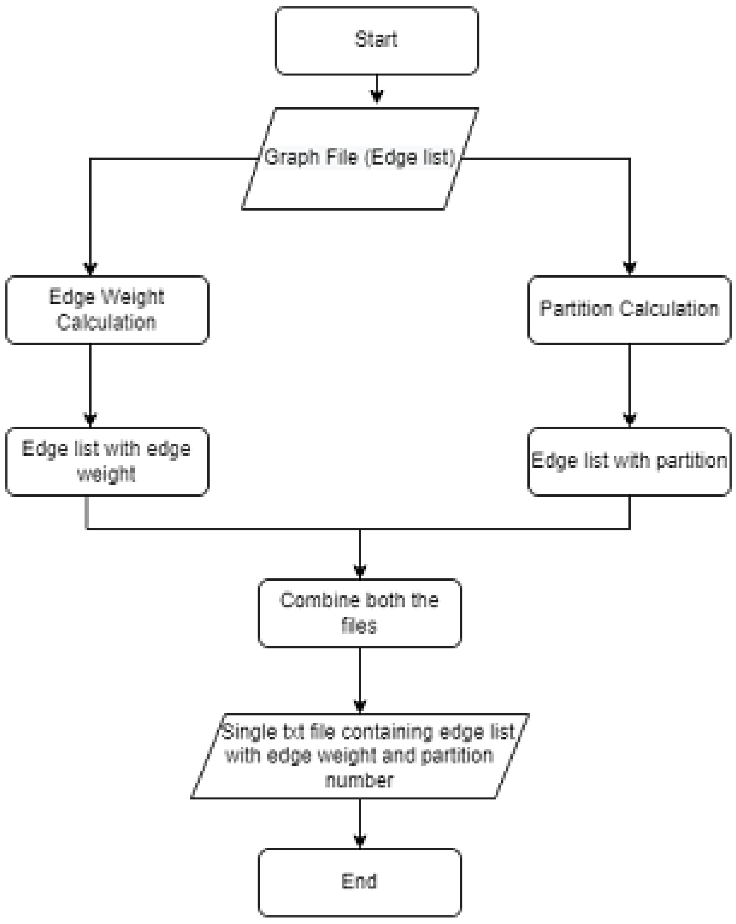


Figure 5.1: Preprocessing step

to other processes. No process is allowed to transmit or receive messages to or from another process throughout a superstep. Any messages sent will only be delivered at the beginning of the next superstep. When all the computations in the superstep are finished and all messages have been dispatched, the superstep comes to a close. At the end of the superstep, a barrier synchronization confirms that all messages have been transferred (but not yet delivered to the processes). Then a new superstep will start and it will deliver the sent messages of the previous superstep to the processes. This process continues until all the processors decide to stop.

5.3 Existing Distributed framework

The existing distributed framework is based on the BSP model. In the distributed framework, every computing node is connected to every other node. Whenever data needs to be sent to another computing node, a new socket connection is created and data is transmitted. A master server is one that is used to perform the synchronization step across all the computing nodes. Once the connection is made, all the computing nodes send Bonjour messages to the master server and wait for the ßtartcommunication"message from the master server. The master server, on receiving the Bonjour messages from all the computing nodes, sends the ßtartcommunication"message to all the other computing nodes. After the initial setup, the compute method is called at every computing node. This method contains an actual algorithm part that provides the implementation for influence spread calculation. A message listener provided in the existing framework is responsible for continuously reading and parsing the messages. It identifies the type of message received by reading the first byte of the message. It delivers the synchronization messages immediately but stores the data message in the message queue. Data messages are delivered only at the beginning of the next superstep.

5.4 Issues in existing Distributed framework

The existing distributed framework was working with only one superstep cycle. It was not able to handle multiple supersteps being performed. Minor changes were made in the messagelistener so that it could run for multiple superstep cycles.

The existing distributed model was working, but it had a few issues. Every time data was sent to another computing node, a new socket was created. This socket creation increased communication time. Also, the messages were sometimes not received in the correct order. To handle this issue, a socket is created once for a connection, and the same socket is used every time for sending data to that computing node.

In the existing distributed framework it was never required to know the size of the message because every time a new socket was created, a message was sent and the socket was closed. But the new changes of a single connection require the message length to be known because the socket is created once and data is continuously sent on this socket connection whenever required. So, in one read on the receiver side, many messages may be included together. To handle this issue, we appended the size of the message as the first 2 bytes before sending the message. On the receiver side, it is known that the first 2 bytes of the message are reserved for the length of the message. As a result, we extract the length from the message by reading two bytes and then reading the message from the buffer until it reaches the length.

The framework allows the sending and receiving of data to and from every computing node to every other computing node. We designate a computing node as the master that is responsible for synchronization using superstep cycles. In the existing framework, every time a new socket was created, it never caused any error in the synchronization of computing nodes while connecting to each other. When the master starts, it starts its listener and waits for the initial bonjour message from all the slave nodes. When it receives the bonjour message from all slaves, it sends the start of communication message, and all the slaves can start the computation. In this, slave nodes connect to other slave nodes only when the exchange of data is required. With the new change of a single socket for each connection, synchronizing computing node connections requires an additional step at the beginning. After the master starts its listener to receive data from all the slaves, it waits for the slaves to connect to the master and get the initial bonjour message. When the master receives the bonjour message from all the slaves, it tries to connect to all the slaves so it can use this connection to send data to all the slaves. The master node sends communication done messages to all the slaves so that they can initiate the connection with the rest of the slaves. Another cycle of bonjour messages and communication done messages are exchanged for this purpose.

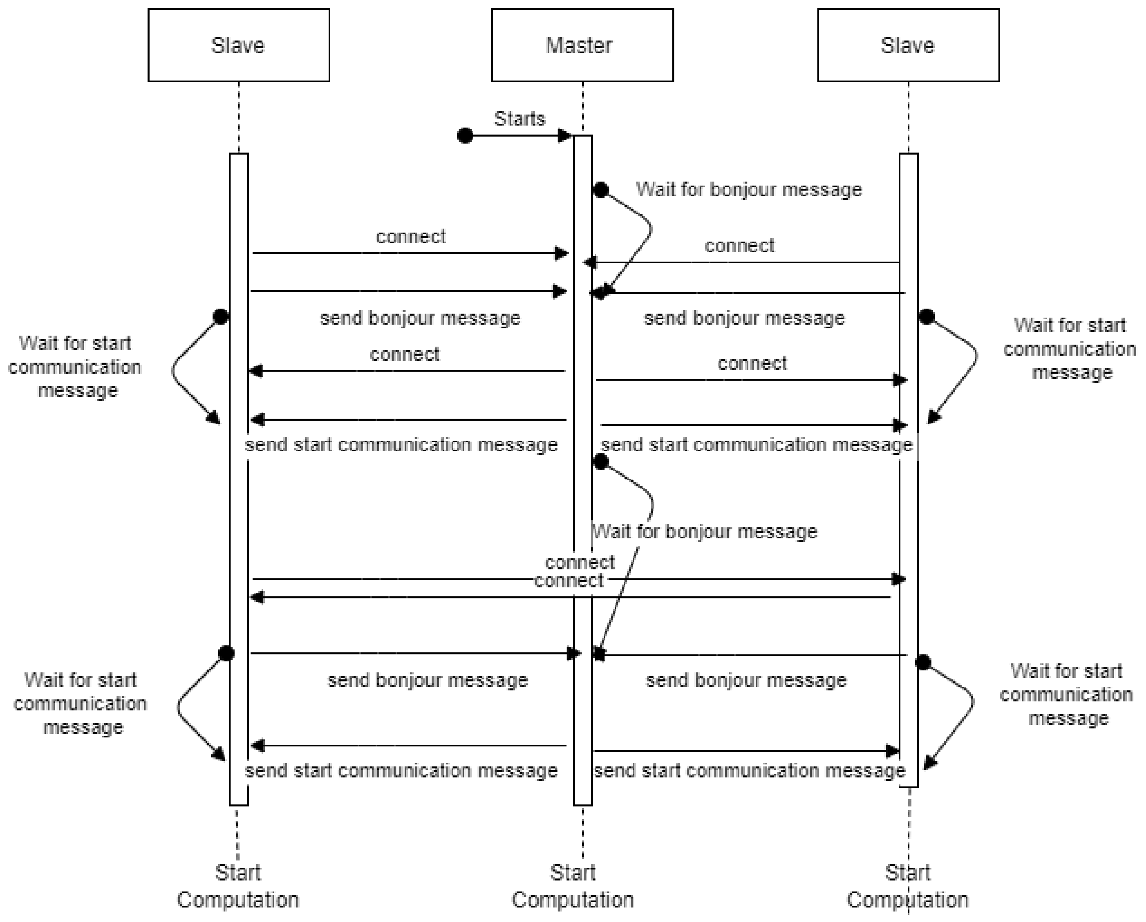

Slave
Master
Slave
Starts
Wait for bonjour message
connect
connect
send bonjour message
send bonjour message
Wait for start communication message
connect
connect
Wait for start communication message
send start communication message
send start communication message
Wait for bonjour message
connect
connect
Wait for start communication message
send bonjour message
send bonjour message
Wait for start communication message
send start communication message
send start communication message
Start Computation
Start Computation
Start Computation

Figure 5.2: Connecting nodes

6 Evalution

In the following section, we investigate the influence spread, runtime, and memory usage of distributed IM algorithms. The first subsection provides an overview of the testing environment and methods. It also explains the graphs that are used for testing the distributed algorithm. The next section first presents a comparison of runtime and influence spread between the centralized and distributed IM algorithms. The second part shows the memory required by the different IM algorithms. In this section, we provide an evalution of distributed UA and distributed IMM algorithm against the state-of-the-art. The final section shows the comparison of all the algorithms we have used in this work.

6.1 Evalution Environment

To evaluate our algorithms, we used two machines. The two machines have two AMD EPYC 7401 24-core processors with 256 GB RAM each. Both machines run Ubuntu 20.04.4 LTS. We have created four partitions of a given graph file. Two instances of the framework work on each machine. The two machines are linked via a 1 gigabit/s connection.

We have considered the weighted cascade model as a diffusion model in our algorithms. This model uses inverse in-degree of nodes. This model assumes that a user is equally likely to adopt information from the users he follows. For example, people have the same trust in the people they follow. We have used three different data sets taken from SNAP [LK14] which are publicly available. The details of the nodes and edges of the graphs can be found in table 6.1. The first one is the ego-Facebook data set, which is comprised of undirected Facebook "circles"(i.e., "friends lists"). This is the smallest dataset that we have used with only 4039 vertices. The second data set is a mid-size dataset, the com-DBLP graph, which represents a comprehensive list of research papers in computer science. If two writers publish at least one paper together, then it creates a co-authorship link connecting them. The last data set is the LiveJournal online social network which is the largest dataset used in our evaluation. A free online blogging community called LiveJournal allows users to express friendship with one another. Users on LiveJournal can also create groups that other users can join. The graph shows LiveJournal friendship. The Live Journal data set is a good candidate for evaluating algorithms for large graphs. It is also used by most of the other IM algorithms for evaluation [JHC12; TSX15].

All the datasets are undirected, as the graph partitioning algorithm GraphPartitioner [1] works with undirected graphs. We have used NE (Neighbor Expansion) [ZWL+17] partition algorithm provided by GraphPartitioner. The partitioning algorithm adds the reverse edge for each existing edge associated with a partition number. Hence, the reverse edge column in Table 6.1 represents edges

[1] https://github.com/zongshenmu/GraphPartitioners

DataSet	Vertices	Edges	Edges(after adding reverse edge)
Fb data	4,039	88,234	1,76,648
Dblp data	3,17,080	1,049,866	2,099,732
com-LiveJournal	3,997,962	34,681,189	34,681,189

Table 6.1: Datasets.

DataSet	Partition 0	Partition 1	Partition 2	Partition 3
Fb data	49,533	42,873	50,731	33,331
Dblp data	7,70,817	5,10,410	4,18,892	3,99,613
com-LiveJournal	15,177,057	7,796,853	5,295,100	6,412,179

Table 6.2: Edges on each partition.

after the partitioning algorithm. In the case of a live journal, the size of the data set increases significantly after adding the reverse edges. Hence, we don't add the reverse edge in the case of a live journal dataset.

In order to understand algorithm performance with different seed set sizes, we have taken the seed set ranges of 10-150 for all the evaluations of algorithms. As distributed algorithms involve communication over a network, we have different runtimes for the same number of seed vertices. Hence, we have run each algorithm 10 times to get the confidence interval of the runtime of different distributed algorithms.

6.2 Evaluation of Distributed Algorithms

The following section shows the experimental result of running the distributed UA and IMM algorithm. Three different measures are being taken into consideration. First, the effectiveness of the algorithm. We calculate the effectiveness of an algorithm in terms of the number of activated nodes at the end of the diffusion process initiated by the computed seed set, i.e., we consider the influence spread value as the effectivity of the algorithm. We try to have the maximum influence spread for a given network. The greater the spread of influence, the more effective the algorithm. Second, we keep track of how long each algorithm takes to compute the desired seed set. Reduced runtime is desirable. We consider the runtime as the efficiency of an algorithm. The lower the runtime, the higher the efficiency of the algorithm.

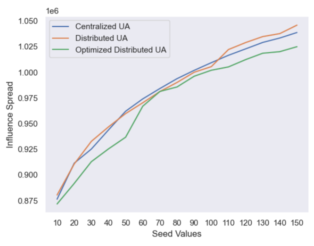
The third measure is the scalability of the algorithm while implementing distributed versions of the influence spread algorithm. It's not always feasible to handle large-scale graphs like live journal entirely on a single instance of an algorithm. In the distributed version of the IM algorithm, we work with partitions of graphs so as to have the scalability of the IM algorithms. In the case of a centralized IM algorithm, the algorithm needs to store and process the entire graph with all its vertices and edges. If we consider a large dataset like Live Journal data, the centralized algorithm needs to process all the 3,997,962 vertices along with all 34,681,189 edges. But if we consider the distributed version of the IM algorithm, these values are reduced as we work with the partitions of graphs. We have run the distributed algorithm with four partitions. Hence, partition zero processes only 15,177,057 edges out of 34,681,189 edges. Table 6.2 represents the number of edges of all partitions for the different data sets.

We measure the scalability of the algorithm in terms of resources required by the algorithm, like memory requirements for running the algorithms. To measure the memory required by the algorithms, we have used the Linux top command [Ker]. We have monitored the resident and virtual sizes by continuously running the top command. Resident (RES) memory refers to the physical memory that the process is using on an operational basis. Virtual (VIRT) presents everything in use and/or reserved by the process at the present [Ker]. We have taken the highest value of RES and VIRT encountered while running the IM algorithms to consider how scalable the IM algorithms are. We do not consider the reduction in runtime as a scalability measure. As we are communicating data over the network, the runtime of distributed algorithms is greater than the runtime of centralized algorithms.

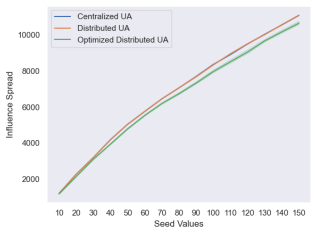
6.2.1 Evaluation of Distributed UA Algorithm

The following section shows the experimental result of running the Distributed UA algorithm. Fig 6.1 shows the comparison of influence spread achieved between the centralized and distributed UA algorithms. As we can see from the plot, the influence spread is almost the same when compared to centralized UA except for the Facebook dataset. Fig 6.1 (b) shows results for the Dblp data set. For the Dblp data set, the distributed UA shows the same performance as that of the centralized UA. We can see that the plot lines for centralized and distributed UA are overlapped. For the Facebook data set, distributed version shows an influence spread little less compared to the centralized UA. In the case of the distributed version, we synchronize the node scoring function values at every step. Hence, the values with which we start the seed selection in the distributed algorithm are the same as those of the centralized algorithm. This results in almost the same influence spread in the case of the distributed version.

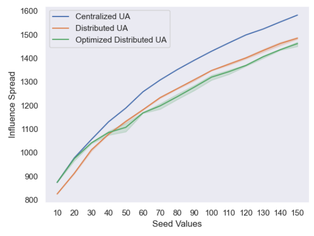
A small discrepancy in influence spread is occasionally observed because we perform forward and backward updates of selected nodes locally and do not synchronize the node scoring function values after these updates. The distributed UA algorithms perform quite well when the marginal gain is not changed much by selecting good vertices. We select the good vertex at the master and send this vertex as a seed vertex to other replicas of it. where they perform a forward update of the vertex, in which it updates the one-hop and two-hop neighbors of the vertex. Then we take the next highest vertex at that partition without any synchronization of values in between and perform the backward update on that vertex. Then we send the next highest vertex at that partition as a candidate seed to the master. Here we are not synchronizing the node scoring function after performing the updates. But we are still getting the good seed vertex, so it is not affecting marginal gain much when we are performing the updates independently on each partition. i.e., removing the influence of the vertex on its neighboring vertices does not induce much difference in marginal gain. Hence, we get almost the same influence spread at the end as that of the centralized algorithm.



(a) Live Journal Dataset

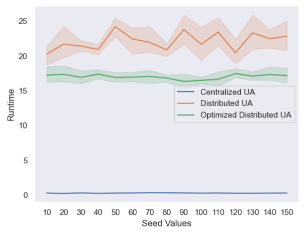


(b) Dblp Dataset

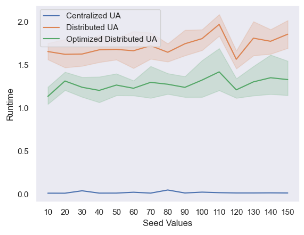


(c) Facebook Dataset

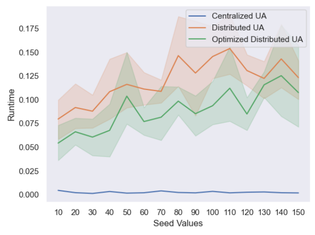
Figure 6.1: Seed vs Influence spread UA



(a) Live Journal Dataset



(b) Dblp Dataset



(c) Facebook Dataset

Figure 6.2: Seed vs Runtime UA

The influence spread in the case of an optimized distributed version is always a little less compared to the centralized and distributed versions of UA. The reason behind this is that we do not synchronize the node scoring function values at each iteration of the algorithm, which adds a little error in the final calculated node scoring values of vertices. We only synchronize the values at the end of the last iteration. Hence, these values differ from the centralized and distributed UA versions.

Fig. 6.2 represents the runtime comparison for the centralized UA, distributed UA, and optimized distributed UA. The runtime for distributed UA is significantly higher compared to centralized UA because of the synchronization of data at each iteration. If a partition has m vertices in $V_{\mathrm{p}}^{\mathrm{ma}}$ and each vertex has an average of r replica servers, then the partition will send m*r messages at every iteration. Here we have run the distributed algorithm with four partitions and three iterations, so approximately 4*3*m*r messages are sent during the synchronization of the node scoring function values. This communication of messages increases the runtime for the distributed UA algorithm. Additionally, each partition performs a synchronization barrier step after sending the value of a seed vertex to the master. Master and other partitions can only use the received data once the master sends confirmation that all the other partitions have finished sending their data. Hence the distribution of the vertices in the partition plays a role here. If one partition has more vertices and edges compared to others it becomes an issue. For example, we are running distributed UA on four partitions. Consider partitions numbered 0,1,2, having 100 vertices and partition numbered 3 having 300 vertices, and assume each vertex are having approximately 2 other replicas. So partitions numbered 0, 1, and 2 will send 100 * 2 messages for each synchronization, and partition 3 will send 300 * 2 messages. Partitions numbered 0, 1, and 2 have to wait for this additional processing and sending of 200 * 2 messages of partition numbered 3 even after done with synchronization of its own vertices. The distribution of edges on different partitions for different datasets can be seen in table 6.2. The distribution of edges is not even on all the partitions. Hence, this impacts the runtime of the algorithm.

In the case of the optimized distributed UA version, we can see that runtime is reduced compared to the distributed version of the UA algorithm. We synchronize the data only at the end of the iterations. So, during the synchronization of the node scoring function values, we roughly send 4*m*r messages irrespective of the number of iterations where m is vertices in ${V_p}^{\mathrm{ma}}$ and each vertex has an average of r replica servers. This helps in reducing the runtime at the cost of a little less influence spread compared to the distributed UA version. As we can see from the graphs, the optimized distributed version can reduce the runtime by up to 30 % of the distributed UA at the cost of a reduction of approximately 2 % in influence spread. We accept this minor reduction in influence spread in the case of the optimized version because the reduction in the runtime of the algorithm is significant.

The third measure we study here is the memory usage of the distributed and centralized algorithms to see the scalability of the algorithms. Tables 6.3 and 6.4 show the resident memory and virtual memory used by different versions of the UA algorithms, respectively. We have monitored the memory usage using the top command. Consider table 6.3, which presents the resident memory usage of the centralized UA algorithm and the memory usage by each partition of the distributed UA algorithm. As we can see, for the Facebook data set and the Dblp dataset, the memory required by each partition is less compared to the centralized algorithms. In the case of the Facebook dataset, the memory reduction is almost 40 % of the centralized UA algorithm. This is because, in the case of the Facebook dataset, the number of edges on each partition is almost the same and makes up approximately 25–30 % of the total number of edges. The number of edges of the dataset on different partitions can be seen in Table 6.2. In the case of the Dblp dataset, the memory is nearly reduced by 30 % of the memory required by centralized UA. Here the memory required by each partition shows a difference because the number of edges of each partition varies significantly and the highest number of edges is on partition zero, hence it shows the highest memory usage compared to other partitions. In the case of live journal data, memory reduction is not that significant. The reason is the number of edges of the live journal data set on each partition differs by a huge amount. Almost 44 % of the total edges of the Live journal dataset is present on partition zero and the rest of the edges are divided into other three partitions. Hence, the highest memory required is 1.9 Gb by partition zero, and the lowest memory required is 1.3 Gb. If we consider the memory required by partition zero, it shows a reduction of 13 % compared to centralized UA. If we assume that each partition will have an approximately similar number of edges, then that count will be around the number of edges present on partition 1. if we take the memory usage of this partition 1 then it shows a reduction of almost 31 % in memory compared to the centralized UA algorithm. The virtual memory required by the different versions of the UA algorithm is shown in Table 6.4, which shows less virtual memory is used by the distributed UA compared to the centralized UA version.

In the case of processing a graph, we need to store additional data to work with the partitions. We calculate and store the main partition value for the vertices that are on a given partition. We also store a replica partition of these vertices. For the vertices that do not belong to the partition, we store details of its replica partition. This is needed by the master server when it sends the forward update of vertices that are not present on its own partition. This use of additional data increases the

DataSet	Centralized UA	Distributed UA				memory reduction in Distributed UA
		partition0	partition1	partition2	partition3	
Facebook	16.7m	9.1m	8.9m	9.8m	9.2m	40%
Dblp	197.6m	139m	118m	109m	106m	30%
Live Journal	2.2g	1.9g	1.5g	1.3g	1.3g	13%

Table 6.3: Resident memory usage for UA.

DataSet	Centralized UA	Distributed UA / Optimized Distributed UA
Facebook	495.6m	299.6m
Dblp	723.0m	418.5m
Live Journal	3,637.6m	2,253.0m

Table 6.4: Virtual memory usage for UA.

memory usage of the distributed algorithms. The way the graph is partitioned affects the runtime and memory usage of the distributed UA algorithm more compared to the influence spread of the distributed UA algorithm.

6.2.2 Evaluation of Distributed IMM Algorithm

In this section, we investigate the performance of the distributed IMM algorithm.

The parameter settings for the distributed IMM version were taken from the existing centralized IMM algorithm [TSX15]. Fig 6.3 shows the comparison of influence spread achieved between the centralized and distributed IMM algorithms. The influence spread comparison of the Live Journal data set is shown in Fig. 6.3(a). As we can see from the plot, the influence spread is less when compared to centralized IMM. In the case of Dblp dataset also we have less spread compared to the centralized version. As we work with partitions of graphs, it might be possible that not all the edges of a vertex are present on the same partition. Because of this, it is possible that even if a vertex is reverse reachable from many other vertices, it is still not present in many RR sets at a single partition. Hence, it has a lower counter value compared to other vertices. This may cause the potential seed vertex to be skipped because of the graph partitioning.

In the case of the distributed version of IMM, we only work with the vertices that belong to the partition while creating RR sets. In the case of a centralized version, we consider all the vertices. Hence, the size of the RR set generated in the centralized and distributed versions differs because of the missing edges in the distributed version. The total number of RR sets generated is lower in centralized IMM compared to distributed IMM. The number of RR sets generated in each iteration when seed size = 140 is shown in tables 6.5, 6.6, and 6.7. The fraction of RR sets covered is used in the stopping criteria for creating a RR set. In the centralized case, as all the edges of vertices are available, it reaches this value early and stops creating RR sets. In the first distributed IMM version, the creation of the RR set is similar to that of the centralized version, where we select only one random vertice. The influence spread achieved in this case is almost 90 % of the influence spread achieved in the case of a centralized algorithm. In the other version, we select 2 random

Iterations	Centralized IMM	Distributed IMM 2nodes	Distributed IMM 2 Nodes and $\frac{\theta}{2.5}$
1	15,744	15,261	6,105
2	31,488	30,521	12,209
3		61,042	24,417
4		1,22,084	48,834

Table 6.5: Live journal dataset RR sets size at partition zero for IMM algorithms

Iterations	Centralized IMM	Distributed IMM 2nodes	Distributed IMM 2 Nodes and $\frac{\theta}{2.5}$
1	12,608	11,527	4,611
2	25,216	23,053	9,222
3	50,431	46,106	18,443
4		92,212	36,885
5		1,84,423	73,770
6		3,68,846	

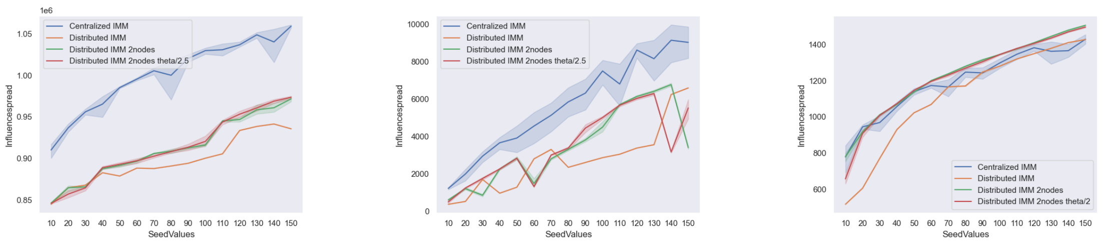
Table 6.6: Dblp dataset RR sets size for IMM algorithms

nodes while creating the RR set. In this case, we achieve a higher influence spread compared to the previous distributed algorithm. The spread of influence, in this case, is almost 92-95 % of the centralized IMM algorithm. The reason behind this is that we select two random nodes while creating the RR sets. By selecting two vertices while creating a single RR set, we are increasing the probability of more vertices being present in different RR sets. As the vertex is present in more RR sets, it will cover a larger number of RR sets. By selecting two random vertices, we basically increase the probability of those vertices appearing in more RR sets.

In the third version, we changed the θ value that is used for creating the number of RR sets. The idea here was to evaluate θ values in such a way that it reduces the number of RR sets generated in each iteration without affecting the influence spread. We evaluated influence spread by reducing the theta value to 2.5, 3, and 4. Out of all this, we found that $\theta = \frac{\theta}{2.5}$ yields a good influence spread compared to the others. After dividing θ by 2.5, it has the same influence spread as when we used θ. Table 6.5 presents the size of the RR sets generated for partition zero during K = 140 for different versions of the IMM algorithm for the Live journal dataset. The case of the centralized algorithm and the distributed IMM algorithm with 2 nodes shows nearly the same RR sets size. But the FR value used in the stopping criteria for generating RR sets is achieved in the second iteration of centralized IMM. Hence, it does not generate any further RR sets. But this is not the case for a distributed IMM with two nodes. Hence it generates RR sets in iterations three and four also. The same behavior can be seen in distributed IMM 2 nodes with $\frac{\theta}{2.5}$ but the advantage here is that the number of RR sets generated in each iteration is reduced. Even with reduced RR sets size the distributed IMM 2 nodes with $\frac{\theta}{2.5}$ provide the same influence spread as that of distributed IMM 2 nodes but it reduces the runtime of the algorithm. Tables 6.6 and 6.7 show the RR sets sizes generated for different versions of the IMM algorithm for the Dblp and Facebook data, respectively.

Iterations	Centralized IMM	Distributed IMM 2nodes	Distributed IMM 2 Nodes and $\frac{\theta}{2.5}$
1	6,087	5,318	2,127
2	12,173	10,635	4,254

Table 6.7: Facebook dataset RR sets size for IMM algorithms



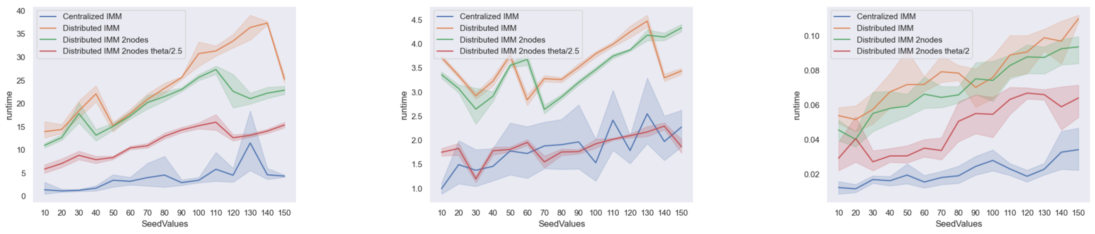
(a) LiveJournal Dataset **(b)** Dblp Dataset **(c)** Fb Dataset

Figure 6.3: Seed vs Influence spread IMM

Fig. 6.4 presents the runtime comparison for the centralized IMM, the first version of the distributed IMM, and the second version of the distributed IMM. The runtime for distributed IMM algorithms is significantly higher compared to centralized IMM because of the generation of a larger number of RR sets. We work with fewer vertices in the case of the distributed version of IMM, but the stopping criteria for creating an RR set include the number of vertices. The number of vertices is always higher for the centralized version of IMM compared to distributed versions of IMM. Also, the fraction of RR sets in R that are covered by vertices in set S differ in both cases. The runtime difference in both versions of distributed IMM is also due to the difference in the number of RR sets generated. As we add 2 nodes in the case of the second version of distributed IMM, a greater number of RR sets are covered from the total RR set generated, which causes early termination of the creation of RR sets compared to the first version of distributed IMM.

In the third version, we take two random vertices and also divide the θ by 2.5 to reduce the RR sets size. In this case, we get the same influence spread as that of the distributed IMM with 2 nodes, but as fewer RR sets are generated, the algorithm becomes faster compared to the previous distributed IMM versions. The distributed IMM with 2 nodes and $\frac{\theta}{2.5}$ shows a nearly 50-60 % reduction in runtime compared to other distributed IMM versions. In the case of the DBLP and Facebook datasets, this version of the distributed IMM shows nearly the same runtime as the centralized IMM version. The runtime of distributed IMM in the percentage of centralized IMM algorithm is significantly less compared to the runtime of distributed UA in the percentage of centralized IMM algorithm. Reduced runtime is the benefit in the case of the distributed IMM algorithm compared to the distributed UA algorithm.

Table 6.8 presents resident memory used by centralized and distributed IMM algorithms. In the distributed version of IMM, we can see a reduction in memory usage by approximately 22 - 29 % of the total memory used by the centralized IMM algorithm. This is because while graph processing we store only edges and vertices that belong to the specific partition. Table 6.9 shows the virtual memory used by the distributed and centralized algorithms. The virtual memory shows a significant increase in the case of distributed IMM versions.



(a) LiveJournal Dataset **(b)** Dblp Dataset **(c)** Fb Dataset

Figure 6.4: Seed vs Runtime IMM

DataSet	Centralized IMM	Distributed IMM				memory reduction in Distributed IMM
		partition0	partition1	partition2	partition3	
Facebook	18m	9.9m	9.6m	10.9m	14m	22%
Dblp	197m	143m	132m	141m	117m	27%
Live Journal	2.4g	1.7g	1.4g	1.3g	1.3g	29%

Table 6.8: Resident memory usage for IMM.

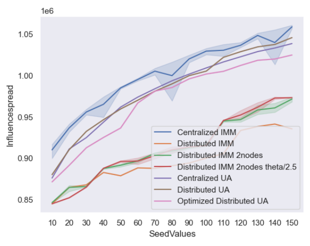
6.2.3 Evaluation of Distributed UA vs Distributed IMM Algorithm

As the final evaluation, we compare all the versions of distributed algorithms and centralized algorithms. Figure 6.5 presents the influence spread vs. seed values for all the algorithms that we have seen. If we consider the influence spread, the distributed UA algorithm performs better compared to all the versions of the distributed IMM in the case of the Live Journal and Dblp data sets. In case of Live journal and Dblp data set the influence speard of Distribuuted IMM versions are approximately 94 % of distributed UA algorithms. In the case of the Facebook dataset, the distributed IMM with 2 nodes and the distributed IMM with 2 nodes with $\theta/2.5$ performed as good as the distributed versions of the UA algorithms.

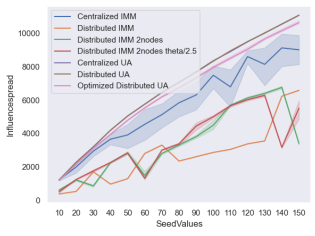
Figure 6.6 shows the runtime comparison of all the algorithms that we studied in this thesis. Distributed IMM version with 2 nodes and $\theta/2.5$ gives the minimum runtime in the case of the live journal and Fb dataset. In the case of the Dblp dataset, optimized distributed UA shows the smallest runtime. Distributed IMM shows the highest runtime in the case of the live journal and

DataSet	Centralized IMM	Distributed IMM
Facebook	495.6m	6697m
Dblp	723.0m	6844m
Live Journal	3,637.6m	8,387m

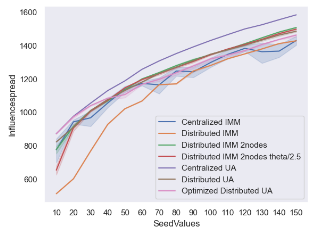
Table 6.9: Virtual memory usage for IMM.



(**a**) LiveJournal Dataset

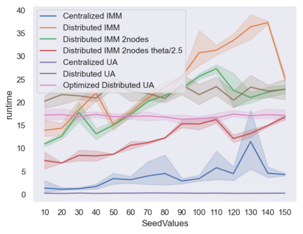


(**b**) Dblp Dataset

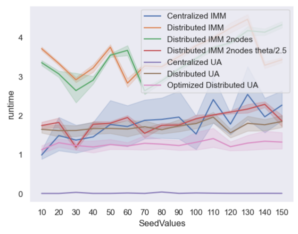


(**c**) Fb Dataset

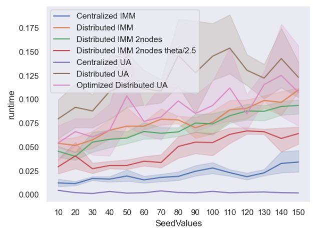
Figure 6.5: Seed vs Influence spread



(**a**) LiveJournal Dataset



(**b**) Dblp Dataset



(**c**) Fb Dataset

Figure 6.6: Seed vs Runtime

Dblp datasets. In the case of the live journal and a Dblp data set, the runtime of the distributed IMM version with two nodes and $\theta/2.5$ is almost half the runtime of the distributed optimized UA algorithm.

In the case of the distributed UA, the runtime is quite constant because we do not have any randomness in the UA algorithm. This is not the case for distributed IMM algorithms. In the distributed IMM case generating of RR sets differs in each run hence we see fluctuations in the runtime of distributed IMM.

Table 6.10 represents the highest resident memory used by all the IM algorithms that we studied in this thesis. The centralized IMM algorithms show the highest memory usage amongst all the algorithms as they generate RR sets. Distributed UA and distributed IMM show a reduction in memory usage between 22 - 30% of the centralized version of UA and IMM, except for the live journal data set in the case of distributed UA.

DataSet	Centralized UA	Centralized IMM	Distributed UA	Distributed IMM
Facebook	16.7m	18m	9.8m	14m
Dblp	197.6m	197m	139m	143m
Live Journal	2.2g	2.4g	1.9g	1.7g

Table 6.10: Highest resident memory usage of IM algorithms.

From all of the above comparisons, we see that in most cases, the runtime of the distributed IMM with 2 nodes and $\theta/2.5$ is less than the distributed UA, and the influence spread of the distributed UA is greater than the distributed IMM. The memory requirement for both the distributed UA and distributed IMM is nearly the same.

7 Conclusion and Future work

In this chapter, we first outline our conclusion based on our previous work. In this section, we show the result of implementing the distributed version of IM algorithms. Next, we recommend some directions for future research and development.

7.1 Conclusion

In this thesis, we have studied the problem of influence maximization on social networks. The concept is as follows: What are the optimal K seeds to choose to maximize the total influence, given that you are aware of the structure of a network and how information is propagated within it? To solve this problem, there are numerous efficient and effective existing algorithms. But these algorithms suffer from scalability issues when working with large-scale graphs. To solve this issue, we implemented two distributed versions of IM algorithms that use less memory per machine compared to centralized IM algorithms. The scaling of the IM algorithm, which is a significant problem in most of the existing algorithms when utilized for larger graphs, will be made easier via distributed IM. While working with a large-scale graph, spread estimation might need a lot of CPU and memory processing power, and it might not be feasible on a single conventional machine. In the case of distributed algorithms, we run multiple instances of the algorithm on different machines, and each instance works with a partition of a graph. In the case of a centralized IM algorithm, the algorithm works with the entire set of vertices and edges of the graph, which can cause a scalability issue when working with large-scale graphs. In the distributed version, we work with a vertex-partitioned graph, so each instance works with fewer vertices than the centralized algorithm. Also, we only store edges that belong to the partition, reducing the overhead of storing other partition edges. Using distributed algorithms to disperse the workload across several computers can help process massive amounts of data.

We implemented a distributed version of the UA algorithm. The distributed version of the UA algorithm gives almost the same influence spread as compared to the centralized version for most of the dataset. To get better influence, we synchronize the vertices' scoring function values at every step of the algorithm. This results in good influence spreading, but at the cost of runtime overhead. The runtime of a distributed algorithm is longer compared to a centralized algorithm as it involves communication over a network. To reduce the runtime, we implemented an optimized version where we reduce the number of times the synchronization of vertices' scoring function values occurs, regardless of the number of iterations. Hence, the optimized version of the distributed algorithm does not show much variation in the runtime. The optimized version shows a reduction in runtime by up to 30 % of the distributed UA at the cost of a reduction of approximately 2 % in influence spread. Further, we studied the scalability of IM algorithms. As the distributed UA version works with partitions of a graph, the memory usage of this distributed UA shows a reduction in memory between 13 % and 40 % compared to the centralized UA for the different datasets. The uneven distribution of edges on different partitions affects the runtime and memory usage of the distributed algorithms.

Another algorithm that we implemented is the distributed IMM algorithm. In the case of distributed IMM, we studied how the FR and θ values affect the RR sets size when working with the partition of graphs. In distributed IMM, we achieve a 90 % influence spread compared to the centralized IMM algorithm. We implemented another version of the distributed IMM algorithm in which we selected 2 random variables to create a RR set to increase the influence spread. The influence spread achieved in this case is almost 92-95 % of the centralized IMM algorithm. But the runtime for these algorithms is higher compared to the centralized IMM version. We found that by reducing the θ value, we could achieve a low runtime value as it reduces the number of the RR sets in most of the datasets without affecting the influence spread that we calculated without decreasing the θ value. Hence, In the third version of distributed IMM, we reduced the θ value by taking θ to $\frac{\theta}{2.5}$. In the case of the distributed version of IMM, we can see a reduction in memory usage by approximately 22–25 % of the total memory used by the centralized IMM algorithm.

7.2 Future work

In this thesis, we have implemented distributed versions of IM algorithms where we work with graph partitions that require storing additional information. This information storage can be investigated to store the least amount of data necessary to manage the graph partitions. This can further help to reduce the memory usage of distributed versions.

In the case of the distributed UA algorithm, the runtime increases because of the synchronization over the network. The master server waits until it receives all the data from other partitions. Hence, if the number of vertices and edges is not distributed evenly to all the partitions then one needs to wait even after it is done with its calculation. In the future, we should explore the graph partitioning algorithm, which provides a consistent proportion of edges and vertices on all the partitions. Also, the graph partitioning algorithm used in the current thesis works only with undirected graphs. Hence, we have used an undirected dataset with added reverse edges to evaluate the algorithms. The algorithm should be evaluated with existing bidirectional datasets. The next thing that can be further investigated is the synchronization delays in the case of a distributed algorithm, which can help reduce the runtime of the algorithm. We are sending separate messages for each vertex over the network. We can check further how the processing gets affected if we combine the number of messages into a single message and send it in a single message.

In the case of distributed IMM algorithms, we have selected two vertices while creating RR sets to increase the influence spread. This can be further evaluated to see how selecting different numbers of random vertices affects the influence spread. The parameter setting plays an important role in generating the RR sets. We have checked the FR and θ values to understand the RR set size. The other parameters, like epsilon, used in IMM must be checked and evaluated for the distributed version of the IMM algorithm which can help to achieve the balance between the number of RR sets to generate and to get good influence spread. Due to the modifications in IMM and the partitioning, we cannot provide any guarantees anymore. The reevaluation of these approximate guarantees is also a valid and relevant point for future work.

In the case of the distributed IMM, an additional algorithm can be explored in which every partition suggests k vertices to the master server. The master server will ask for details of all these K* (number of the partition) vertices from other partitions that did not suggest these vertices. Basically, it will ask for the counter values for these vertices from other replicas. The master will combine all the counter values from all the partitions, find the maximum counter value vertex, and add it to the seed set. We expect a quality gain due to the additional information about a vertex received from other partitions.

Bibliography

[BBCL14] C. Borgs, M. Brautbar, J. Chayes, B. Lucier. “Maximizing Social Influence in Nearly Optimal Time”. In: SODA ’14. Portland, Oregon: Society for Industrial and Applied Mathematics, 2014, pp. 946–957. ISBN: 9781611973389 (cit. on pp. 19, 23).

[CDPW14] E. Cohen, D. Delling, T. Pajor, R. F. Werneck. “Sketch-Based Influence Maximization and Computation: Scaling up with Guarantees”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. CIKM ’14. Shanghai, China: Association for Computing Machinery, 2014, pp. 629–638. ISBN: 9781450325981. DOI: 10.1145/2661829.2662077. URL: https://doi.org/10.1145/2661829.2662077 (cit. on p. 22).

[CLC13] W. Chen, L. V. S. Lakshmanan, C. Castillo. *Information and Influence Propagation in Social Networks*. Morgan amp; Claypool Publishers, 2013. ISBN: 1627051155 (cit. on p. 16).

[CPL12] Y.-C. Chen, W.-C. Peng, S.-Y. Lee. “Efficient Algorithms for Influence Maximization in Social Networks”. In: *Knowl. Inf. Syst.* 33.3 (Dec. 2012), pp. 577–601. ISSN: 0219-1377. DOI: 10.1007/s10115-012-0540-7. URL: https://doi.org/10.1007/s10115-012-0540-7 (cit. on p. 13).

[CWW10] W. Chen, C. Wang, Y. Wang. “Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks”. In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’10. Washington, DC, USA: Association for Computing Machinery, 2010, pp. 1029–1038. ISBN: 9781450300551. DOI: 10.1145/1835804.1835934. URL: https://doi.org/10.1145/1835804.1835934 (cit. on p. 22).

[CWY09] W. Chen, Y. Wang, S. Yang. “Efficient Influence Maximization in Social Networks”. In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’09. Paris, France: Association for Computing Machinery, 2009, pp. 199–208. ISBN: 9781605584959. DOI: 10.1145/1557019.1557047. URL: https://doi.org/10.1145/1557019.1557047 (cit. on p. 21).

[CYZ10] W. Chen, Y. Yuan, L. Zhang. “Scalable Influence Maximization in Social Networks under the Linear Threshold Model”. In: *Proceedings of the 2010 IEEE International Conference on Data Mining*. ICDM ’10. USA: IEEE Computer Society, 2010, pp. 88–97. ISBN: 9780769542560. DOI: 10.1109/ICDM.2010.118. URL: https://doi.org/10.1109/ICDM.2010.118 (cit. on p. 22).

[DR01] P. Domingos, M. Richardson. “Mining the Network Value of Customers”. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’01. San Francisco, California: Association for Computing Machinery, 2001, pp. 57–66. ISBN: 158113391X. DOI: 10.1145/502512.502525. URL: https://doi.org/10.1145/502512.502525 (cit. on p. 21).

[GAR16] S. Galhotra, A. Arora, S. Roy. “Holistic Influence Maximization: Combining Scalability and Efficiency with Opinion-Aware Models”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 743–758. ISBN: 9781450335317. DOI: 10.1145/2882903.2882929. URL: https://doi.org/10.1145/2882903.2882929 (cit. on p. 23).

[GBR21] H. Geppert, S. Bhowmik, K. Rothermel. “Large-Scale Influence Maximization with the Influence Maximization Benchmarker Suite”. In: GRADES-NDA ’21. Virtual Event, China: Association for Computing Machinery, 2021. ISBN: 9781450384773. DOI: 10.1145/3461837.3464510. URL: https://doi.org/10.1145/3461837.3464510 (cit. on pp. 14, 18, 23, 25, 26).

[GLL11a] A. Goyal, W. Lu, L. V. S. Lakshmanan. “SIMPATH: An Efficient Algorithm for Influence Maximization under the Linear Threshold Model”. In: *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*. ICDM ’11. USA: IEEE Computer Society, 2011, pp. 211–220. ISBN: 9780769544083. DOI: 10.1109/ICDM.2011.132. URL: https://doi.org/10.1109/ICDM.2011.132 (cit. on p. 22).

[GLL11b] A. Goyal, W. Lu, L. V. Lakshmanan. “CELF++: Optimizing the Greedy Algorithm for Influence Maximization in Social Networks”. In: *Proceedings of the 20th International Conference Companion on World Wide Web*. WWW ’11. Hyderabad, India: Association for Computing Machinery, 2011, pp. 47–48. ISBN: 9781450306379. DOI: 10.1145/1963192.1963217. URL: https://doi.org/10.1145/1963192.1963217 (cit. on p. 21).

[IBC10] D. Ienco, F. Bonchi, C. Castillo. “The Meme Ranking Problem: Maximizing Microblogging Virality”. In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. ICDMW ’10. USA: IEEE Computer Society, 2010, pp. 328–335. ISBN: 9780769542577. DOI: 10.1109/ICDMW.2010.127. URL: https://doi.org/10.1109/ICDMW.2010.127 (cit. on p. 13).

[JHC12] K. Jung, W. Heo, W. Chen. “IRIE: Scalable and Robust Influence Maximization in Social Networks”. In: *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*. ICDM ’12. USA: IEEE Computer Society, 2012, pp. 918–923. ISBN: 9780769549057. DOI: 10.1109/ICDM.2012.79. URL: https://doi.org/10.1109/ICDM.2012.79 (cit. on pp. 23, 35).

[Ker] M. Kerrisk. *Linux Top Man page*. URL: https://man7.org/linux/man-pages/man1/top.1.html (visited on 10/01/2022) (cit. on p. 37).

[KKT03] D. Kempe, J. Kleinberg, É. Tardos. “Maximizing the Spread of Influence through a Social Network”. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’03. Washington, D.C.: Association for Computing Machinery, 2003, pp. 137–146. ISBN: 1581137370. DOI: 10.1145/956750.956769. URL: https://doi.org/10.1145/956750.956769 (cit. on pp. 13, 17, 18, 21).

[KS06] M. Kimura, K. Saito. “Tractable Models for Information Diffusion in Social Networks”. In: *Proceedings of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases*. ECMLPKDD’06. Berlin, Germany: Springer-Verlag, 2006, pp. 259–271. ISBN: 3540453741 (cit. on p. 22).

[LK14] J. Leskovec, A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014. (Visited on 09/01/2022) (cit. on pp. 31, 35).

[LKG+07] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, N. Glance. “Cost-Effective Outbreak Detection in Networks”. In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’07. San Jose, California, USA: Association for Computing Machinery, 2007, pp. 420–429. ISBN: 9781595936097. DOI: 10.1145/1281192.1281239. URL: https://doi.org/10.1145/1281192.1281239 (cit. on pp. 13, 21).

[LXC+14] Q. Liu, B. Xiang, E. Chen, H. Xiong, F. Tang, J. X. Yu. “Influence Maximization over Large-Scale Social Networks: A Bounded Linear Approach”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. CIKM ’14. Shanghai, China: Association for Computing Machinery, 2014, pp. 171–180. ISBN: 9781450325981. DOI: 10.1145/2661829.2662009. URL: https://doi.org/10.1145/2661829.2662009 (cit. on p. 23).

[LZT15] Y. Li, D. Zhang, K.-L. Tan. “Real-Time Targeted Influence Maximization for Online Advertisements”. In: *Proc. VLDB Endow.* 8.10 (June 2015), pp. 1070–1081. ISSN: 2150-8097. DOI: 10.14778/2794367.2794376. URL: https://doi.org/10.14778/2794367.2794376 (cit. on p. 13).

[OSFK17] N. Ohsaka, T. Sonobe, S. Fujita, K.-i. Kawarabayashi. “Coarsening Massive Influence Networks for Scalable Diffusion Analysis”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 635–650. ISBN: 9781450341974. DOI: 10.1145/3035918.3064045. URL: https://doi.org/10.1145/3035918.3064045 (cit. on p. 14).

[PQD+15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni. “HDRF: Stream-Based Partitioning for Power-Law Graphs”. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM ’15. Melbourne, Australia: Association for Computing Machinery, 2015, pp. 243–252. ISBN: 9781450337946. DOI: 10.1145/2806416.2806424. URL: https://doi.org/10.1145/2806416.2806424 (cit. on p. 31).

[SSVS22] S. S. Singh, D. Srivastva, M. Verma, J. Singh. “Influence Maximization Frameworks, Performance, Challenges and Directions on Social Network: A Theoretical Study”. In: *J. King Saud Univ. Comput. Inf. Sci.* 34.9 (Oct. 2022), pp. 7570–7603. ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2021.08.009. URL: https://doi.org/10.1016/j.jksuci.2021.08.009 (cit. on pp. 15–17, 21).

[Staa] Statista. *Statista*. URL: https://www.statista.com/statistics/303681/twitter-users-worldwide/ (visited on 09/01/2022) (cit. on p. 13).

[Stab] Statista. *Statista*. URL: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/ (visited on 09/01/2022) (cit. on p. 13).

[STLS06] X. Song, B. L. Tseng, C.-Y. Lin, M.-T. Sun. “Personalized Recommendation Driven by Information Flow”. In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’06.

Seattle, Washington, USA: Association for Computing Machinery, 2006, pp. 509–516. ISBN: 1595933697. DOI: 10.1145/1148170.1148258. URL: https://doi.org/10.1145/1148170.1148258 (cit. on pp. 13, 18).

[TSX15] Y. Tang, Y. Shi, X. Xiao. “Influence Maximization in Near-Linear Time: A Martingale Approach”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1539–1554. ISBN: 9781450327589. DOI: 10.1145/2723372.2723734. URL: https://doi.org/10.1145/2723372.2723734 (cit. on pp. 14, 23, 28, 29, 35, 40).

[TXS14] Y. Tang, X. Xiao, Y. Shi. “Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 75–86. ISBN: 9781450323765. DOI: 10.1145/2588555.2593670. URL: https://doi.org/10.1145/2588555.2593670 (cit. on p. 21).

[Val11] L. G. Valiant. “A Bridging Model for Multi-Core Computing”. In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 154–166. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2010.06.012. URL: https://doi.org/10.1016/j.jcss.2010.06.012 (cit. on pp. 25, 31).

[WLJH10] J. Weng, E.-P. Lim, J. Jiang, Q. He. “TwitterRank: Finding Topic-Sensitive Influential Twitterers”. In: *Proceedings of the Third ACM International Conference on Web Search and Data Mining*. WSDM ’10. New York, New York, USA: Association for Computing Machinery, 2010, pp. 261–270. ISBN: 9781605588896. DOI: 10.1145/1718487.1718520. URL: https://doi.org/10.1145/1718487.1718520 (cit. on p. 13).

[YKK13] H. Yu, S.-K. Kim, J. Kim. “Scalable and Parallelizable Processing of Influence Maximization for Large-Scale Social Networks?” In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE ’13. USA: IEEE Computer Society, 2013, pp. 266–277. ISBN: 9781467349093. DOI: 10.1109/ICDE.2013.6544831. URL: https://doi.org/10.1109/ICDE.2013.6544831 (cit. on p. 22).

[ZLH14] Z. Zong, B. Li, C. Hu. “Dirier: Distributed influence maximization in social network”. In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2014, pp. 119–125 (cit. on p. 23).

[ZWL+17] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, Z. Li. “Graph Edge Partitioning via Neighborhood Heuristic”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’17. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 605–614. ISBN: 9781450348874. DOI: 10.1145/3097983.3098033. URL: https://doi.org/10.1145/3097983.3098033 (cit. on pp. 31, 35).

Declaration