SAGE: A System for Uncertain Network Analysis

Eunjae Lee*
LINE+, Korea
eunjae@linecorp.com

Sam H. Noh
UNIST, Korea
samhnoh@unist.ac.kr

Jiwon Seo†
Hanyang University, Korea
seojiwon@hanyang.ac.kr

ABSTRACT

We propose SAGE, a system for uncertain network analysis. Algorithms for uncertain network analysis require large amounts of memory and computing resources as they sample a large number of network instances and run analysis on them. SAGE makes uncertain network analysis simple and efficient. By extending the edge-centric programming model, SAGE makes writing sampling-based analysis algorithms as simple as writing conventional graph algorithms in Pregel-like systems. Moreover, SAGE proposes four optimization techniques, namely, deterministic sampling, hybrid gathering, schedule-aware caching, and copy-on-write attributes, that exploit common properties of uncertain network analysis. Extensive evaluation of SAGE with eight algorithms on six real-world networks shows that the four optimizations in SAGE jointly improve performance by up to 13.9× and on average 2.7×.

1 INTRODUCTION

Networks are hard to understand because of their complex structures. What makes them even harder to understand is their uncertainty. Many networks that are of interest to us have uncertain structures or probabilistic connections. For example, biological networks such as protein interaction networks [63] are constructed from experimental observations, thus their connectivity may be erroneous or probabilistic [84]. In mobile ad-hoc networks, mobile devices may travel and connect or disconnect from each other, which may be modeled as uncertain networks [5, 15]. Figure 1(a) shows an example uncertain network with edge existence probabilities and five sample networks (b–f) derived from this network.

To make sense of uncertain networks, probabilistic and approximate analysis algorithms have been studied [6, 13, 14, 19, 25, 28, 29, 32, 38, 41, 46, 48, 51, 52, 60–63, 68, 76, 80, 81, 87, 90]. While these algorithms differ in detail, many of them share a common structure of 1) sampling network instances, 2) running analysis on the sampled networks, and 3) aggregating the analysis results [6, 13, 14, 19, 29, 32, 41, 46, 48, 51, 62, 63, 68, 76, 80, 87]. Sampling networks and running their analyses cause significant overhead of storing the sampled networks and performing computations on all these networks. Thus, writing efficient analysis algorithms for uncertain networks requires considerable engineering effort. To ease this difficulty, we propose Sampling-Aware Graph Engine, or SAGE, a system and programming model for uncertain network analysis.

While existing systems such as Pregel [55] help with large-scale network analysis, making use of them for uncertain networks is by no means straightforward. Their programming models do not support network sampling nor are they efficient in handling sampled networks in terms of storage and computations. With

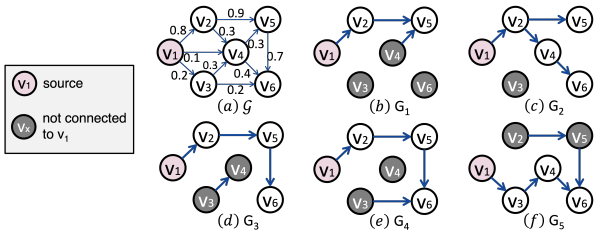


Figure 1: Example uncertain network (a) and its sample networks (b–f); the numbers are the edge existence probabilities.

the availability of uncertain networks in many application domains [18, 20, 31, 72, 73, 78, 84], their analysis is becoming increasingly important. Despite this, the lack of a simple means for expressing the problems has hindered their use. We propose SAGE that makes uncertain network analysis simple and efficient. Its high-level programming constructs help to express uncertain network algorithms, which are then accelerated with the optimizations provided in SAGE. The contribution of this paper is as follows.

A System and Programming model. To the best of our knowledge, SAGE is the first system with a programming model that supports uncertain network analysis. The programming model, i.e., data and computation model, is designed to express all existing sampling-based algorithms that we reviewed. Moreover, we carefully design the programming model so that SAGE can incorporate the domain-specific optimizations that we propose. As the optimizations significantly reduce storage and computation overhead, they are essential to making uncertain network analysis practical.

Domain-specific system optimizations. The main sources of overhead in uncertain network analysis are three-fold: 1) the memory and storage overhead of maintaining millions of sampled networks, 2) the computation overhead of processing those large number of sample networks, and 3) the overhead of storing and accessing a large amount of vertex attributes of all sample networks. We propose four optimizations to address these problems, namely, deterministic sampling, hybrid gathering, schedule-aware caching, and copy-on-write vertex attributes that take advantage of the common properties of uncertain network analysis. For example, deterministic sampling mitigates the memory and storage overhead of maintaining sampled networks by dynamically and deterministically sampling the existence of edges, while schedule-aware caching reduces the amount of I/O to access vertex attributes.

Extensive evaluation and availability. All the optimizations presented in this paper have been implemented in our prototype system. Eight core algorithms for uncertain network analysis are succinctly implemented in SAGE. We evaluate the algorithms with six real-world networks – three social networks, one biological network, and two computer application networks. Our optimizations

*Work done while at UNIST
†Corresponding author and principal investigator

improve overall performance by a maximum 13.9×, and 2.7× on average, while reducing the memory usage to 23.7% on average over the baseline that represents a state-of-the-art uncertain network analysis system. We open sourced the entire system as well as the experimental settings and results in our repository [43].

The rest is organized as follows. Section 2 gives the background on the algorithms and systems for uncertain network analysis. Section 3 presents Sage's programming model and Section 4 describes its design and architecture with a focus on our optimizations. Section 5 evaluates the Sage prototype system and Section 6 concludes.

2 BACKGROUND AND RELATED WORK

Algorithms for uncertain networks have different compute characteristics from conventional network analyses. For uncertain networks, the simple method of regarding uncertainties (or probabilities) as edge weights and running conventional analysis algorithms does not work. That is, when aggregating the existence probabilities of multiple paths leading to a vertex, we must consider their joint probability, which is extremely expensive to compute for large graphs. Moreover, most conventional network algorithms consider edge weights or their aggregation (i.e., vertex attributes) to be independent, and thus we cannot transform edge probabilities to edge weights and apply conventional network algorithms. As such, to design a programming model for uncertain network analysis, we reviewed existing algorithms and studied their characteristics in detail. We find that most of the algorithms are based on randomly sampling networks and iteratively aggregating the results [6, 13, 14, 19, 29, 32, 41, 46, 48, 51, 62, 63, 68, 76, 80, 87]. Here we report the results of our review of core algorithms, that is, top-k reliability search, distance computation, k-nearest neighbors, and k-core decomposition, for uncertain network analysis. Then we review other related work on uncertain network analysis.

Top-k reliability search. Reliability computation is a fundamental problem in uncertain network analysis [25, 32, 63, 77, 87]. Top-k reliability search is used in protein interaction networks to find the proteins that are biologically related to a given protein [10, 63, 87] or in peer-to-peer file-sharing networks to select reliable peers for file transfer [39]. Reliability computes the probability that a source vertex s is connected to other vertices. Top-k reliability search finds k vertices with the connections of the highest reliabilities (or probabilities) from a source vertex s. More formally, for an uncertain network $\mathcal{G}$ and a vertex s in $\mathcal{G}$, reliability of the connection from s to a vertex v is denoted as $R_{\mathcal{G}}(s,v)$ and computed as follows [87]:

$$R_{\mathcal{G}}(s,v) = \sum_{G \in \Omega(\mathcal{G})} P(\mathcal{G} \Rightarrow G) R_G(s,v)$$

where G is a possible network instance, or a sample network, that is obtained by randomly retaining the edges of $\mathcal{G}$ with each edge's probability and Ω is the set of all possible sample networks. $R_G(s,v)$ is 1 if s can reach v in G and 0 otherwise. $P(\mathcal{G} \Rightarrow G)$ is the probability of obtaining G from the random sampling. If the probabilities of all the edges in $\mathcal{G}$ are mutually independent, $P(\mathcal{G} \Rightarrow G)$ is computed as the product of the probabilities of the edges in G and the complement of the probabilities of the edges not in G. For example, in Figure 1, G_1 is a sample network of the uncertain network $\mathcal{G}$. If the existence probabilities of all edges are independent, the probability $P(\mathcal{G} \Rightarrow G_1)$ is the product of the existence probabilities of the edges in G_1 ($V_1 \rightarrow V_2$, $V_2 \rightarrow V_5$, and $V_4 \rightarrow V_5$) and the complement probabilities of the other edges in $\mathcal{G}$.

As top-k reliability search is #P-complete, a sampling-based algorithm is used [87]. That is, N possible networks G_1, ..., G_N are sampled from $\mathcal{G}$ and on each network G_i, we compute $R_{G_i}(s,v)$ for $v \in G_i$. Then, $R_{\mathcal{G}}(s,v)$ is estimated to be $\frac{1}{N}\sum_{G_i} R_{G_i}(s,v)$ and k vertices with the highest estimated $R_{\mathcal{G}}(s,v)$ are selected. For example, consider the network $\mathcal{G}$ in Figure 1 again. Estimated in the way described above, V_2, V_5, and V_6 in Figure 1 are the top 3 reliable vertices with reliability 0.8. For V_3 and V_4, their reliability is 0.2 as one possible network out of five has connected paths to the vertices.

Path distances and nearest neighbors. Computing distances from a source vertex and finding the k-nearest neighbors are important primitives for uncertain network analysis. They are core operations for link prediction, clustering, and graph mining [2, 4, 11, 49, 63]. With probabilistic edge connections in uncertain networks, distances are often measured by median, majority, or most probable distances in the sampled networks [63].

The k-nearest neighbors algorithm computes path distances from a source vertex to find k vertices with the shortest distances. In uncertain networks, the approach is similar to reliability search where N possible networks are sampled and the distances are computed on each sample network. Then, the distances are aggregated to compute the median, majority, or most probable distance for each vertex to find the k nearest vertices. An optimized approach where each possible network is incrementally sampled is possible [63]. That is, we first compute the distances of vertices that are within τ proximity of the source vertex by sampling the edges within this proximity. The computed distances are aggregated to test if all k-nearest neighbors are identified. If more paths need to be explored, the value of τ is incremented, and in the next iteration more edges are sampled in each partially sampled network. This pattern of partial sampling and aggregating analysis results is common in uncertain network analysis [28, 41].

Other uncertain network algorithms. Finding dense subgraphs is an important primitive in uncertain network analysis. In particular, k-core decomposition, which finds the maximal subgraph of vertices having k or more neighbors, has been widely studied for uncertain networks [6, 62]. K-core decomposition may be used to find maximal cliques [12] or to compute influence maximization [13].

Aside from the graph analytic algorithms we have discussed so far, graph mining is another class of graph problem that has been well studied [24, 30, 79]. Graph mining finds matching subgraphs to a given query graph to enumerate the matches or count them. Graph mining algorithm for uncertain networks is an important problem on its own and have also been studied [9, 25, 28, 52, 91]. However, as exemplified by Arabesque [74], a graph mining system proposed separately from graph analysis systems such as Pregel, graph mining systems requires a different set of programming constructs and optimizations. Hence, we leave the support for graph mining algorithms as future work.

Related graph processing systems. Programming models and systems for network analysis have been extensively studied [8, 17, 21, 35, 37, 42, 53–56, 58, 59, 64, 66, 67, 69, 83, 85, 88]. However, we are not aware of any programming model developed exclusively for uncertain networks. Also, we are aware of only one (incomplete) system by Zou at el. [89] developed for uncertain network analysis. Their system implements one optimization for uncertain network

```
class Vertex {
  virtual void Init();            // initialize vertex attributes.
  virtual void Scatter();         // send messages to neighbors.
  virtual void Gather(Vertex src, GEdge e);}; // read src and update this Vertex.
class GVertex {
  virtual void Reduce(Vertex v);// reduction of Vertex v in all sample networks
                                // for this GVertex instance.
  virtual void ReduceDone();};  // invoked when the reduction is done.
class Global {
  virtual void Reduce(GVertex gv); // reduction of all GVertex instances.
  virtual void ReduceDone();};     // invoked when the reduction is done.
```

Figure 2: Vertex API in SAGE. `Vertex` is a vertex in a sample network and `GVertex` is a vertex in an uncertain network.

analysis that reduces redundant vertex computations of sampled networks; we describe the optimization and its limitation as well as our solution in Section 4.3. However, they do not have a proper programming model for uncertain network analysis and thus, they evaluate their prototype with conventional graph algorithms by simply running them for all sampled networks. In reality, uncertain network algorithms cannot be implemented in this way as they require vertex-wise aggregation over all sample networks, possibly multiple times, during their executions. In contrast, SAGE is a complete system with a proper programming model, domain-specific optimizations, and programming APIs.

3 PROGRAMMING MODEL IN SAGE

In this and the following sections, we present the core components of SAGE, the programming model and system for uncertain network analysis that we propose. We first describe its programming model.

We design SAGE's programming model by extending the asynchronous edge-centric computation model [67]. We made this design decision based on two principles that we felt imperative. First, the programming model should be able to express existing uncertain network algorithms. Second, the model should be able incorporate the optimizations that we propose, which are essential in reducing the analysis overhead, thus, making them practical. Regarding the former, upon careful review of existing algorithms, we observed that the analysis of individual sampled networks can be expressed in the edge-centric model. Thus, we extended our programming model to allow aggregation of sampled networks' analysis results (as shown in Section 3.1). We confirmed (by reviewing analysis algorithms and implementing them in SAGE) that by repeating the analysis and aggregation, the programming model can express all the core uncertain network algorithms that we reviewed and evaluated.

Regarding the latter, our programming model and execution mechanism (described in Section 4.1) make implementation of the domain-specific optimizations possible. More specifically, first, the edge-centric model allows to separately process the messages for each incoming edge of a vertex, while the vertex-centric model requires the grouping of all messages and delivering them at once. Hence, we can process all sample networks' messages for the same edge altogether. This largely improves the locality of vertex access, which is exploited in our vertex cache (explained in Section 4.4), and the opportunity to eliminate redundant computations among the sample networks, which is exploited in our hybrid gathering optimization (explained in Section 4.3). Second, by exploiting the asynchronism in our programming model, we do not materialize the messages between vertices, which relieves the massive overhead of creating and storing all sample networks' messages for all vertices. These optimizations (and others) are explained in Section 4.

Algorithm 1: Semantics of SAGE programs

```
1  while active vertex exists do
2      foreach s ∈ SampleNetworks do
3          foreach v ∈ s.ActiveVertices() do
4              v.Scatter()
5      foreach s ∈ SampleNetworks do
6          foreach u → v ∈ s.Edges() do
7              if u triggered v in u.Scatter() then
8                  v.Gather(u, u → v)
9      if active vertex not exists then
10         foreach gv ∈ All GlobalVertices do
11             foreach v ∈ gv.SampleVertices() do
12                 gv.Reduce(v)
13             gv.ReduceDone()
14         if active vertex not exists then
15             foreach gv ∈ All GlobalVertices do
16                 Global.Reduce(gv)
17             Global.ReduceDone()
```

In the rest of the section, we first describe our vertex data model. Then, we use top-k reliability search as an example for describing our programming model. The rest of the algorithms that we surveyed and implemented are included in our source code repository [43].

3.1 Vertex Data Model

SAGE provides two vertex interfaces, `Vertex` and `GVertex` as shown in Figure 2. SAGE programs need to subclass the two classes to implement the analysis algorithms. (Note that the keyword *virtual* requires the functions to be overridden.) The `GVertex` class represents the vertices in the uncertain network and the `Vertex` class represents those in the sample networks. The `Vertex` class is used for running analysis on each sample network in a message-passing manner. The `Scatter()` and `Gather()` functions implemented in the analysis programs are invoked by SAGE for the `Vertex` instances that are active and that have received messages, respectively. `Scatter()` may send messages to neighbor vertices and `Gather()` receives the messages to update vertex attributes. Their semantics are similar to those of conventional graph systems and are described in Section 3.2 in detail. In `GVertex`, the analysis results are aggregated in `Reduce()`, which is invoked individually for all sample networks' vertices for the `GVertex` instance. Then `Reduce()` in the `Global` class performs global aggregation over all `GVertex` instances.

Algorithm 1 describes the high-level semantics of SAGE programs. In each iteration (lines 1–17) SAGE runs a single step of the edge-centric computation for all sample networks, i.e., sending messages over the edges of the networks in `Scatter()` (lines 2–4) and processing the messages to update vertex attributes in `Gather()` (lines 5–8). After this step, if no vertices are active SAGE performs the reductions in lines 9–17. First, vertex-wise reduction of the analysis results in all sample networks is performed (lines 10–13). After this reduction, if no active vertex exists, network-wise reduction for all vertices in the uncertain network is done (lines 14–17). Note that reductions may activate some of the inactive vertices in the sample networks, for which edge-centric computation is executed in the following iteration. If there are no active vertices after the two reductions are completed, the analysis terminates.

```
class TopkVertex: Vertex {
  bool visited = false;
  void Scatter() {
      if (visited == false) {
          visited = true; TriggerGatherOnNeighbors(); }}
  void Gather(Vertex src, GEdge edge) { ActivateSelf(); } };
class TopkGVertex: GVertex {
  int pathcount;
  void Reduce(Vertex v) { if (v.visited) pathcount++; } };
class TopkGlobal: Global {
  PriorityQueue<K> topkVertices; // storing K vertices with highest path count.
  void Reduce(GVertex gv) { topKVertices.Insert(gv, gv.pathcount); } };
```

Figure 3: Top-k reliability search in Sage.

3.2 An Example: Top-k Reliability Search

Top-k reliability search, shown in Section 2, is implemented as sampling networks, running BFS (breadth first search) from the source on each sample network, and counting for each vertex the number of sample networks with connected paths. Then, it finds the k vertices with the highest path counts. We use the algorithm as an example to show the workings of our programming model.

In Sage, top-k reliability search is implemented as in Figure 3. `TopkVertex` implements BFS that is executed for each sampled network. First, the `visited` flags of all vertices in the sample networks are initialized to `false`. Then the source vertex is activated (not shown in the figure) and its `Scatter()` is invoked. In `Scatter()`, the flag of the vertex is set and its neighbor vertices are triggered for `Gather()`, where triggering is similar to sending messages in typical edge-centric systems, except that the messages are not explicitly created but the neighbors are only "triggered". In `Gather()`, we simply activate the target vertex of gather with `ActivateSelf()`, so that the vertex becomes active in the next iteration and `Scatter()` is called for this vertex. The invocation of `Scatter()` and `Gather()` is repeated until all vertices of all sample networks become inactive. When all vertices are inactive, `Reduce()` in `TopkGVertex` is called to count the total number of sample networks with connected paths for each `GVertex` instance. Thereafter, `Reduce()` in `TopkGlobal` finds the vertices (`GVertex` instances) with the k highest count values as the top-k reliable connections.

As in Pregel, Sage programs terminate when all vertices (in all sample networks) are inactive after an iteration. In `Gather()`, `ActivateSelf()` may be called to activate the vertices of the sample networks in the next iteration. Also, in the reduce functions, i.e., `GVertex::Reduce()` or `Global::Reduce()`, we may call `Activate(GVertex)` to activate the vertex in all the sample networks (though this is not used in the top-k reliability search example). In addition to monitoring the activation status of each vertex, Sage internally keeps track of the *triggering* vertices that called `TriggerGatherOnNeighbors()` in `Scatter()` and the *triggered* vertices for which `Gather()` needs to be called in each iteration. The use of these information is described in detail in Section 4.3.

3.3 Limitation of Sage's Programming Model

Our programming model can express a large subset of uncertain network algorithms that are based on randomly generating sample networks and analyzing those sample networks [6, 13, 14, 19, 29, 32, 41, 46, 48, 51, 62, 63, 68, 76, 80, 87]. However, there are four categories of algorithms that cannot be expressed with our programming model. They are 1) algorithms that do not use sampling for the analysis [81, 90], 2) algorithms that find and analyze with a small number of representative sample networks [60, 61] (for which Sage's API cannot help find the representative networks and the optimizations will have only limited effect due to its small number), 3) analysis of networks that have (cyclic) conditional probabilities of edge existences [33], and 4) graph mining algorithms that find subgraphs with certain conditions. As we described in Section 2, graph mining requires a different set of programming constructs from those of graph analysis, we leave the support for graph mining algorithms as future work. For the third one (conditional probability), we need to consider the joint probability of those edges to obtain a sample network by applying sophisticated sampling algorithms such as Gibbs sampling. If a network does not have a cycle, we can sort the edges topologically and sample the edges in that order, which may be supported in Sage with limited parallelism.

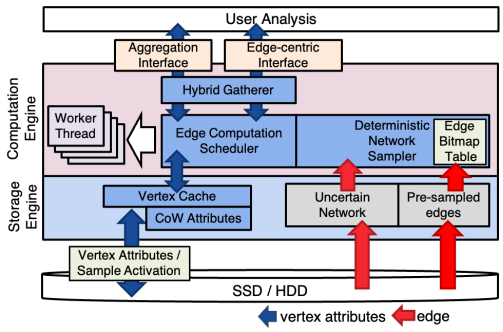


Figure 4: High-level architecture of Sage

Sage adopts the edge-centric computation model, which is used in many graph processing systems [8, 66, 67]. It is as expressive as the vertex-centric model as the edge-centric model can explicitly aggregate the vertex messages to emulate the vertex-centric computation [27]. The asynchronism in our programming model (i.e. updates by gather operations may be arbitrarily ordered) does not limit its expressiveness; Sage programs can store two versions of vertex attributes for the current and previous supersteps and explicitly maintain those values if synchronous semantics is needed.

4 DESIGN AND ARCHITECTURE OF SAGE

The overall architecture of Sage is shown in Figure 4. The system largely consists of two parts, the computation engine and the storage engine. When a user program starts, the storage engine loads the uncertain network from disk to memory. The storage engine also caches and manages the vertex attributes of `GVertex` and `Vertex` as well as the activation status stored in disk. The computation engine provides two programming interfaces, namely, the edge-centric interface and the aggregation interface, with which Sage interacts with user programs to run the analysis algorithms.

Within Sage, we propose four novel optimization techniques for efficient analysis, namely, deterministic sampling, hybrid gathering, schedule-aware caching, and copy-on-write attributes. We first describe the inner-workings of the edge computations and their scheduling in Sage. Then we describe the four optimizations.

4.1 Edge Computations

In each iteration (i.e., *superstep*), conventional graph systems enumerate active vertices and run scatter on them to send messages along the edges. The messages are then gathered on the target

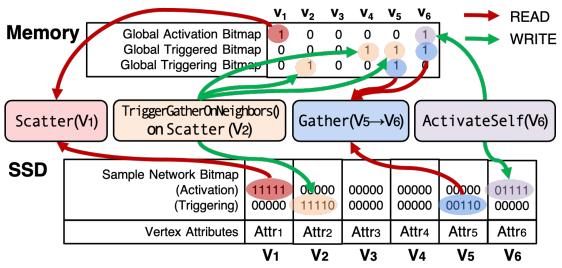


Figure 5: Bookkeeping of vertex status in SAGE. Vertex bitmaps (top), sample network bitmaps (bottom), and the operations using the bitmaps (middle) are shown.

vertices. The messages and activation status of current and next supersteps are stored in bitmaps or other data structures.

SAGE runs the edge computations in a similar way but with a few differences in order to correctly and efficiently execute the computations for a large number of sample networks. This requires maintaining activation status of all vertices for all sample networks, both of which may be in the few to many million range. Also, as the messages between vertices are not materialized in SAGE, it needs to keep track of the sender and receiver vertices in all sample networks to correctly support the edge-centric computation semantics.

For the edge computations, SAGE enumerates the vertices in the *global active vertex set*, which is the set of vertices that are active in *any* sample network, and invokes the scatter on them. For each enumerated vertex, scatter is invoked exactly for the sample networks where the vertex is active. In the scatter function, SAGE simply *triggers* the neighbor vertices that the gather needs to be performed on, rather than actually creating and sending messages. As scatter and gather are performed for all sample networks, sending messages for all those networks in the conventional manner requires a large memory space. When processing gather on the triggered vertices, SAGE reads the attributes of the source vertex of the gather that is to be performed. For the gather operation, SAGE enumerates the vertices in the *global triggered vertex set*, which is the set of vertices being triggered in any sample network, and invokes the gather on them. As with scatter, for each enumerated vertex, gather is invoked for the sample networks where the vertex is triggered.

As described above, SAGE needs to maintain the global vertex status as well as the per-sample vertex status for efficient processing of edge computation. The global vertex status is stored in DRAM as its size is proportional to the number of vertices in the uncertain network and thus requires a small amount of memory (maximum 2.5MB for the networks in Table 4). The per-sample vertex status is stored in SSD as its size is proportional to the number of all vertices in all samples and thus large in size (maximum 920GB for the COG dataset with 500,000 samples in Table 3). SAGE maintains three in-memory data structures for global vertex status: Global Activation Bitmap, Global Triggered Bitmap, and Global Triggering Bitmap. Also, SAGE stores on disk for each vertex its status in the sample networks – its activation status and triggering status (whether or not the vertex triggered gather in the corresponding sample network) that are stored as the Sample Network Activation Bitmap and the Sample Network Triggering Bitmap, respectively.

Figure 5 shows the vertex status for running the reliability search on the five sample networks in Figure 1. It shows on top and bottom, respectively, the global vertex status in memory and the per-sample vertex status on disk, while the middle bubbles show the operations that are making use of the status bitmaps. It shows the operations and activation status, in part, of the first three iterations; scatter for superstep 1, trigger gathering for superstep 2, and gather/activation for superstep 3. Note that the Sample Network Bitmaps for a vertex are stored together with the vertex attributes as the bitmaps and the attributes are accessed at the same time.

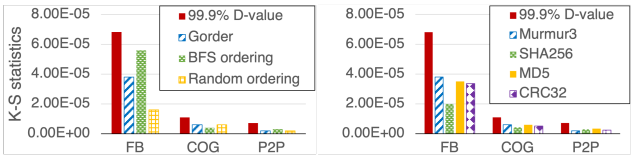
Scheduling of Edge Computations. The edge computations, i.e., the scatter and gather computations, are scheduled by the Edge Computation Scheduler component (EScheduler, for short) in Figure 4 and executed by worker threads. Scatter computations are scheduled to execute and complete before gather operations execute. Both scatter and gather are executed in the order of the vertex IDs. Scatter operations are ordered by the IDs of the scattering vertices, while gather operations are first ordered by the target vertices and then by the source vertices. As the operations are executed in ascending order of vertex IDs, the vertices that will be accessed in the future can be quite accurately estimated. This is not a coincidence because we purposely designed the scheduling of the edge computation to be coordinated with the efficient vertex caching method that we propose, which is executed in the Vertex Cache component in Figure 4 and described in detail in Section 4.4. That is, the computations are scheduled such that the future references of vertex attributes are accurately predicted.

Having described the basic inner-workings of SAGE, we now discuss the systemic challenges in uncertain network analysis and our optimizations for them. In uncertain network analysis, the main sources of overhead are 1) the memory and storage overhead of maintaining sampled networks (e.g. terabytes of memory/storage for millions of sample networks), 2) the computation overhead of processing a large number of sample networks (e.g. trillions of gather/scatter operations in each superstep for one million sample networks and one million edges), and 3) the overhead of storing and accessing a large amount of intermediate data, i.e., vertex attributes of all sample networks (e.g. tens of terabytes of vertex attributes for one million vertices and one million sample networks).

We propose four optimizations to address these problems – deterministic sampling for the memory/storage overhead of materializing sample networks, hybrid gathering for the computation overhead of massive number of sample networks, and efficient caching (schedule-aware caching and copy-on-write attributes) for the overhead of storing/accessing all sample networks' vertex attributes. The following sections describe these four optimizations.

4.2 Deterministic Network Sampling

In uncertain network analysis, generating sample networks is an important part of the analysis. Few studies have considered the issue of materializing sample networks. A few have proposed ways to generate samples, but never discuss the storage overhead involved [26, 29, 57, 82]. The lone existing (incomplete) system for uncertain networks materializes all generated samples [89]. This certainly incurs large memory overhead requiring, for example, 18 terabytes of memory space for the P2P network (described in Table 2) with one million sample networks.



(a) K-S test for graph orderings **(b) K-S test for hash functions**

Figure 6: Statistical testing of deterministic sampling: results show D-statistic smaller than the threshold (red bar) meaning that the samples are identical to uniform random samples.

To alleviate this overhead in Sage, we propose a technique called *deterministic network sampling*, or simply, *deterministic sampling*, that regenerates the sample networks, in effect, replacing memory and storage overhead with slight CPU overhead. The key idea is to exploit the property of pseudorandom number generators that deterministically generate the same sequence of numbers when given the same seed value. This is implemented in the Deterministic Network Sampler component in Figure 4. Consider the sequence of sample networks generated in Figure 1. For each graph, the existence of an edge is determined by the random number generated for the edge. Take for example, the edge between vertices v_1 and v_2, whose value is 0.8. If the random value generated for this edge is below 0.8, the edge exists, as in G_1; if the random value is above 0.8, the edge is disconnected, as in G_5.

In deterministic sampling, we do not generate and maintain all the samples, but check the edges using a pseudorandom number generator whose seed is set to $H(s) + H(t)$, for any edge $s \rightarrow t$, where s and t are the source and target vertex IDs, respectively, and $H()$ is a uniform hash function. Since a pseudorandom number generator with the same seed always generates the same sequence of random values, we can easily check any edge of any sample graph. For example, to check the existence of an edge between vertices v_3 and v_4 for G_3, we simply check the third random value of the pseudorandom number generator with the seed of $H(3) + H(4)$.

The key benefit of this technique is that it saves memory and storage space. The space required is independent of the number of sample networks allowing us to use a large number of samples. The downside of the approach is the CPU overhead for random number generation. However, we find that deterministic sampling is generally far more efficient as we show later with the experiments.

While deterministic sampling helps to minimize the use of memory and storage, we need to verify whether it is statistically sound. That is, we need to test if our deterministic random number generation has the same probability distribution as a pseudorandom number generator. Thus, we conduct a two-sample Kolmogorov–Smirnov (K-S) test [40], which is used to test whether the underlying probability distributions for two sets of samples differ.

To this end, we generate 1,000 random numbers for the edges of a network in the following two ways. First, we use the conventional method of generating random values from one seed for all edges. Second, we use our deterministic sampling method and generate random values for each edge using $H(s) + H(t)$ as the seed value. We repeat the random number generation for three real-world graphs (the characteristics of the graphs are described in detail in Section 5), each with three different graph orderings (i.e. vertex ID assignment) [44, 75] and four hash functions for $H()$. Then, we apply the K-S test for the generated values and compute Kolmogorov's D statistic value for the 99.9% confidence level for each of the tests, of which the results are shown in Figure 6. (We report only a subset of the results in the interest of space, but the results not plotted here show similar trends.) The results clearly show that the computed Kolmogorov's D statistic is far smaller than the threshold (denoted by the 99.9% red bar), indicating that the samples are drawn from identical distributions with 99.9% confidence level.

Algorithm 2: Hybrid Gathering

Input: S, T, e, B_e, R, and N. S and T are the attributes of source (s) and target (t), e is the edge between s and t, B_e is e's existence bitmap for sample networks, R is s's triggering bitmap for sample networks, and N is the number of samples.
Output: T. T is target vertex's attributes.

```
1 if AttrType(S) = AttrBitmapTbl then
2     if AttrType(T) = AttrBitmapTbl then
3         T = CollectiveGather(S, T, e, B_e, R);
4         if Size(T) > AttrSize * N then
5             T = ConvertToArray(T)
6         return T;
7 if AttrType(S) = AttrBitmapTbl then S=ConvertToArray(S) ;
8 if AttrType(T) = AttrBitmapTbl then T=ConvertToArray(T) ;
9 return T = IterativeGather(S, T, e, B_e, R, N)
```

4.3 Hybrid Gathering

We now present our optimization for the edge computations described in the previous section. Recall that sampling occurs from the same uncertain network. Thus, many sample networks have similar structures and many of their edges execute the exact same computations during the superstep computations. For example, in the sample networks of Figure 1, four of the five sample networks G_1, G_2, G_3, and G_4 have a similar structure in that they all have a common path $V_1 \rightarrow V_2 \rightarrow V_5$. Thus, running BFS on these graphs executes the same computations for vertices V_1, V_2, and V_5.

Previously, Zou et al. [89] proposed to eliminate redundant edge computations to improve the performance of uncertain network analysis. Their technique maintains the mappings of an attribute to a bitmap for each vertex, where the *attribute* is that of the vertex and the *bitmap* represents the sample networks sharing the attribute value of the vertex. Then the edge computations for the same attribute pairs of the source and target are computed once *collectively* and the attribute mapping is updated accordingly. We call this *collective gathering* as opposed to the conventional method of iteratively running gather operations for sample networks (which we call *iterative gathering*). The pseudocode of collective and iterative gather is shown in the supplementary material.

Collective gathering reduces the amount of computation for analyses where a small number of distinct values are used for each vertex, as in top-k reliability search. However, for analyses that generate a relatively large number of distinct values per vertex, collective gather may incur significant performance overhead (up to 10× slowdown over iterative gathering in our evaluation) as the size of the attribute-bitmap tables may become large. Since it is difficult to estimate the number of vertex attribute values for an analysis algorithm beforehand, it is difficult to determine whether collective gathering should be used or not for better performance.

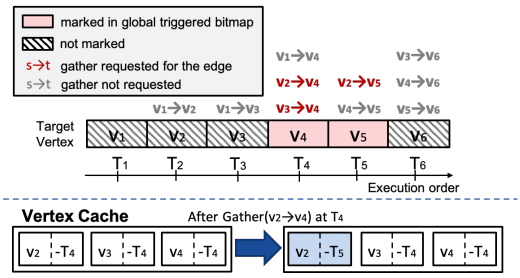


Figure 7: An example scheduling of gather operations. The priority update after running `Gather`($v_2 \rightarrow v_4$) is shown.

In SAGE, we propose a hybrid approach that combines the best of the two gathering methods. Our technique, namely, *hybrid gathering*, selectively applies the cheaper of the two for each edge. That is, if the number of attribute values for a gather operation is larger than a threshold, SAGE applies iterative gathering; otherwise, collective gathering is used. For hybrid gathering, SAGE maintains vertex attributes either as an attribute-bitmap table for collective gather or as an attribute array for iterative gather. As the number of attributes for a vertex generally increases as analysis proceeds, we initially store the attributes in the attribute-bitmap table, but then switch to the array representation once we find that the attribute-bitmap table of the vertex is larger than its array representation.

Algorithm 2 describes hybrid gather in SAGE. If the attributes of the source and target are stored in the attribute-bitmap table, we apply collective gather (lines 1–3). After the gather, if the attribute-bitmap table is larger than the corresponding attribute array (the threshold), we convert it to the array representation (lines 4–5). We apply iterative gather if either the source or target stores the attributes in the attribute array (line 9), after converting the attribute-bitmap table to an array temporarily for the source (line 7) or permanently for the target (line 8). We find that hybrid gather is as fast as the faster of the two gathering methods.

Although hybrid gathering may seem simple and straightforward, dynamically determining the gathering method for individual vertices is not feasible without SAGE's programming model and its efficient execution mechanism described in Section 4.1. For example, if the vertex computation model is used, the messages sent by source vertices must be aggregated before being delivered to the target vertices. Thus, even if the source and target vertices are stored in a bitmap representation, the aggregated message may not be. This forces the target vertices to be converted to the array representation, which makes hybrid gathering ineffective. Moreover, if the messages sent between vertices are materialized (unlike in SAGE), the overhead of conversion will be much higher because multiple messages created by a vertex need to be converted. In contrast, in SAGE, the converted attributes for a vertex can be stored in the cache and used for multiple gather operations. Hence, hybrid gathering is made feasible because of the efficient edge computation in SAGE that does not require message materialization.

4.4 Vertex Cache

To execute the scatter and gather operations, the attributes of a vertex on disk need to be retrieved to the vertex cache in main memory. For scatter, the attributes of the active vertices are accessed exactly once and thus those vertices are sequentially retrieved to the cache and then evicted. For gather, however, the attributes of a vertex may be accessed multiple times. For example, after running `Gather`($s \rightarrow t$), the source s can be accessed later as a source or target of another gather and target t can be a source of another gather. If s or t are accessed later, we want to keep them in the cache so that they need not be retrieved again later. Naturally, as the size of the vertex cache is limited, we need to prioritize the vertices and retain those that are accessed in the immediate future rather than those that are accessed in the distant future. An accurate prediction of future accesses will improve the cache hit ratio and the overall performance. In SAGE, the Vertex Cache component in Figure 4 manages the cache. It takes advantage of the orderly execution of the edge computations by EScheduler, who orders the scatter and gather operations by the vertex IDs. Thus, future accesses of vertex attributes are accurately predictable. Our schedule-aware caching policy estimates the future accesses of vertices, which are encoded and stored as priority scores in cache entries.

Algorithm 3: Priority Update after Gather Operation

```
1 Input: the source s and target t of gather, Output: updated priority of s.
2 if s.nbrLeastLarger(t) exist then
3     if isTriggered(s) and t < s then
4         s.priority = −min(s, s.nbrLeastLarger(t))
5     else s.priority = −s.nbrLeastLarger(t);
6 else
7     if isTriggered(s) and t < s then s.priority = −s ;
8     else s.priority = −inf;
```

The upper figure in Figure 7 shows the execution of gather operations of the second superstep for the sample networks in Figure 1, where the x-axis represents logical time and where at T_i gather operations for target vertex i are executed. The lower figure shows the vertex cache before and after executing `Gather`($v_2 \rightarrow v_4$) at logical time T_4. Before running the gather operation, our vertex cache stores the attributes of v_2, v_3, and v_4, and their priority $-T_4$. The priority values represent the logical time points when the vertices are accessed; e.g., v_3 is accessed at time T_4 for `Gather`($v_3 \rightarrow v_4$) and thus its priority is set to $-T_4$. After running `Gather`($v_2 \rightarrow v_4$), the priorities of v_2 and v_4 are re-computed so that they represent the future access time of the vertices. Since v_2 is accessed at time T_5 as the source of `Gather`($v_2 \rightarrow v_5$), its priority is updated to $-T_5$.

More generally, the priorities are computed in the following manner. Consider the gathering of s to t ($s \rightarrow t$) executed at time T_t (which equals t). If vertex s or t is fetched to the cache at that point, their priority is set to $-T_t$ (or $-t$). After the gather is performed, the priorities of vertex s and t are updated according to the estimation of their next access time. Let us first consider the source of the gather. Vertex s may be accessed as a target of a future gather if s is larger than t; s may be a source of another gather as well. Algorithm 3 considers these cases and correctly updates the priority of s after `Gather`($s \rightarrow t$). If s has an outgoing neighbor whose ID is larger than t ($s.nbrLeastLarger(t)$), then s may be accessed as a source of another gather at logical time $s.nbrLeastLarger(t)$ (line 2–4). If s is triggered and $t<s$ then s will also be accessed as a target of a gather at logical time T_s. The minimum of the two estimated access times (s and $s.nbrLeastLarger(t)$) is used to update s's priority (line 3). If s is accessed only as a target of another gather, its priority is either set to $-s$ or $-infinity$ (line 6–7). To efficiently find the smallest

Table 1: Evaluated Algorithms

Algorithm	Description
Top-k reliability search (**TopK**)	Finds the k most reliably connected vertices from a given source vertex as shown in Figure 3 [87].
K-nearest neighbors (**kNN**)	Finds k vertices that are closest to a given vertex by incrementally running Dijkstra's algorithm [63].
Personalized PageRank (**PPR**)	Computes the importance of vertices for a given source vertex. The scores for each vertex in sample networks are averaged.
K-core decomposition (**kCore**)	Finds the maximal subgraph consisting of the vertices with degree K or larger. Our implementation computes probabilistic (k, η) cores [6].
Influence maximization (**IM**)	Finds the vertex with the highest influence for a given set of vertices based on the LT (linear threshold) model.
Breadth-First Search (**BFS**)	Finds the median number of hops from a source vertex.
Shortest paths (**SP**)	Finds the median distance of each vertex from a given source vertex.
Network clustering (**CL**)	Finds densely-connected clusters of vertices by running the reliability search algorithm and recursively dividing the network [28, 41].

Table 2: Summary of Evaluated Networks

Network	$\|V\|$	$\|E\|$	Domain	Vertex Attr. Size
Facebook (**FB**)	63.7K	817K	Social Network	31MB–99GB
Youtube (**YT**)	1.13M	2.99M	Social Network	554MB–1.7TB
Skitter (**SKT**)	1.70M	11.09M	Computer Network	828MB–2.6TB
COG (**COG**)	223K	31.41M	Bio Network	109MB–345GB
Orkut (**ORK**)	3.07M	117.2M	Social Network	1.5GB–4.7TB
eDonkey P2P (**P2P**)	5.8M	147.8M	Computer Network	2.8GB–9.0TB

neighbor larger than t, we precompute such a vertex for the target t of each edge $s \rightarrow t$ and store the vertex with the edge data.

Now consider the target vertex t. If t triggered other vertices in the current superstep, it may be accessed as a source of future gather. If so, we search t's neighbor vertices to find the one that is triggered in the current superstep and has the (smallest) ID larger than t's. If such vertex v exists, then we set t's priority to be $-v$; if not, we set its priority to be $-infinity$.

If a single worker thread is used, the gather operations are totally ordered as shown in the upper figure of Figure 7. In this case, our scheme accurately predicts the vertex accesses. In contrast, if multiple worker threads execute gather operations, the execution may not be totally ordered by the vertex IDs. While this makes vertex access less predictable, our evaluation shows that our caching policy improves the hit ratio by an average and maximum of 6 and 18 percentage points, respectively, compared to the LRU scheme.

Note that for gather operations within one superstep, our caching policy is (almost) identical to Belady's algorithm of evicting those that are accessed *furthest in the future* with the highest priority [3], which is proven optimal. When we use multiple worker threads for gather operations, our policy does not perform optimal eviction, which is a trade-off between parallelism and cache performance.

4.5 Copy-on-Write Attributes

Many analyses start by setting the attributes of all vertices to be the same value. As analyses proceed, some vertices update their attributes but others may retain the initial value for many iterations, even until the end of the analysis. In k-nearest neighbors, the distances of all vertices are initially set to infinity and many of them still hold infinity distance after the k neighbors are identified.

Sage exploits this common initial attribute of vertices to reduce the storage overhead and improve performance by storing the initial value once and letting all the vertices share the common initial value. Then Sage applies copy-on-write when the attributes of the vertices are updated. This optimization is automatically applied if the `Init()` function with no argument in `Vertex` class is overridden. Our evaluation shows that this optimization is effective for traversal-based algorithms such as top-k reliability search.

5 EVALUATION

This section shows the evaluation of our Sage prototype system with eight core algorithms for uncertain network analysis. We first describe the evaluation settings. Then we show the performance gains brought by the overall and individual optimizations in Sage.

5.1 Algorithms and Datasets

We implemented a prototype of Sage with all the optimizations in Section 4. To evaluate Sage, we use eight uncertain network algorithms, namely, top-k reliability search (TopK), k-nearest neighbors (kNN), Personalized PageRank (PPR), k-core decomposition (kCore), influence maximization (IM), Breadth-First Search (BFS), shortest paths (SP), and network clustering (CL), as summarized in Table 1. These were chosen as they are representative sampling-based algorithms that efficiently perform their respective analysis. For the TopK, kNN, and kCore computation, the evaluated algorithms are, to the best of our knowledge, the most efficient and general algorithms [6, 34, 63, 87]. Aside from the algorithms we chose, we are also aware of other efficient algorithms that take a slightly different approach. For TopK, the algorithm by Khan et al. [32] builds an offline index (which can be costly to compute for multiple source vertices) to efficiently process online queries. This online processing part runs a sampling-based reliability search (similar to our implementation) and thus, can be expressed in Sage. For kCore, there is an index-based implementation [76, 80] to efficiently answer online queries for all k and η (probability threshold) values. However, this implementation has different applications than the general kCore computation [6].

For PPR, the one we implement is considered the definitive version although there are a few approximate algorithms [14, 38]. The IM problem may be computed based on either the IC (Independent Cascade) model or the LT (Linear Threshold) model [31]. The former does not require a sampling-based method but the latter does (for counting influenced neighbor vertices and deciding the influence status of a vertex). Our IM implementation represents a subset of the IM algorithms that are based on the LT model [31]. BFS and SP are similar to TopK and kNN with k set to infinity. The evaluated CL algorithm is the most scalable one that can run on networks with millions of vertices [7, 19, 50].

We evaluate the algorithms with six real-world networks in the public domain [45, 65, 71] as shown in Table 2. Four of these, YT, SKT, COG, and ORK, were used in earlier uncertain network studies [23, 76, 80, 89],while FB and P2P are from domains where uncertain network analysis is commonly applied. They represent networks from three domains, specifically, FB, YT, and ORK are friendship networks from the social network domain, while COG is a biological network representing interactions between proteins. SKT and P2P are computer networks, with SKT being an Internet routing network and P2P a peer-to-peer file-sharing network. Note that the size of these networks are similar to those that are used in

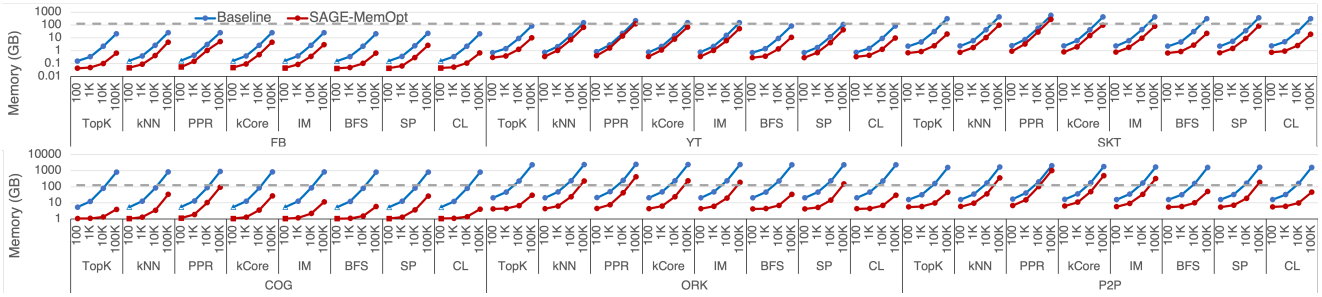


Figure 8: Memory usage of Sage-MemOpt and the baseline system. The gray horizontal dotted line indicates the size of DRAM on the evaluated machine. The plots above this line are estimated (i.e., manually computed) values.

the majority of previous studies on uncertain network analysis [22, 23, 26, 32, 33, 48, 57, 62, 63, 76, 80, 86, 87, 89].

Of these networks, FB and COG contain data to calculate the edge probabilities. Specifically, FB contains the frequency of the communications between two connected vertices. COG has the confidence scores for the protein interactions. For these two networks, we estimated the edge existence probabilities from the given data and use them in our experiments. For the others, we synthetically generate the probabilities with uniform random distribution. In addition, as two algorithms, kNN and SP, require edge lengths we generate edge lengths in the range of 1∼100 with Zipfian distribution where longer lengths have higher probabilities. Throughout the experiments with these networks, we make use of 100 to 100,000 sample networks in 10× increments, which are the numbers used in the majority of existing studies [1, 14, 22, 26, 32, 33, 47, 57, 62, 63, 68, 86, 87, 89]. The size of the total vertex attributes with 100–100,000 sample networks for the algorithms (when our optimizations are not applied) are shown in the table. Aside from these experiments, in Section 5.3.6 we consider the scalability of Sage, where we experiment with larger synthetic networks of 100 million to 2 billion edges while considering 100,000 to 5 million sample networks. Note that the largest synthetic network in our evaluation are 10× larger than the largest network evaluated in existing uncertain network studies [32, 52]. Also the maximum sample size in our evaluation (5M) is 5× larger than the largest sample size (1M) in previous studies [47].

5.2 Hardware and System Settings

For all the evaluations, we use a machine with Intel Xeon E5-2690 v4 running at 2.6GHz with fourteen cores. The machine has 128GB DRAM and a 1TB NVMe SSD (Samsung 970 PRO). As we strictly control the size of the vertex cache, the memory used for running the experiments is much smaller than 128GB. We report the actual memory usage measured with `GNU time` [16]. We use fourteen threads that process both the computation and I/O.

We evaluate Sage and our four proposed optimizations – deterministic sampling, hybrid gathering, schedule-aware caching, and copy-on-write attributes. Because the optimizations affect both the memory footprint and the execution times of the algorithms, we evaluate Sage with three settings, namely, baseline system, Sage-MemOpt, and Sage-ExecOpt. They are configured as the following.

Baseline system: This is Sage without the four optimizations. The baseline system stores materialized sample networks in main memory in compressed form. That is, the existence of each edge for the sample networks is encoded as a single bit in a bitmap, and the bitmaps for all the edges are kept in memory. The baseline sets the vertex cache size to 5% of the total vertex attribute size of all sample networks. The cache applies LRU eviction policy.

Sage-MemOpt: This is Sage with all four optimizations. With Sage-MemOpt, we focus on evaluating the amount of memory that Sage can optimize. We set its vertex cache size to be the same as the baseline (i.e., 5% of the total vertex attributes), but deterministic sampling largely reduces the memory usage of Sage-MemOpt.

Sage-ExecOpt: This is Sage with all four optimizations and with the same memory usage as the baseline. With Sage-ExecOpt, we evaluate the performance gain of Sage when given the same amount of memory as the baseline. To level memory consumption with the baseline, Sage-ExecOpt first increases the vertex cache size. If the memory consumed is still lower than the baseline even with the maximum cache size (i.e., the size of the total vertex attributes), Sage-ExecOpt materializes the sample networks for a subset of vertices for which their edge existence bitmaps are stored in memory.

5.3 Evaluation Results

In discussion the results, we first justify the performance of the baseline system and then compare its evaluation results with that of Sage-MemOpt and Sage-ExecOpt.

To demonstrate the efficiency of the baseline system, we compare its performance with existing graph processing systems – PowerGraph [17], X-Stream [67], and Ligra [70]. As comparison with existing systems is not the main focus of our evaluation, we do not describe the details of the results. However, we do want to point out two facts from the results. First, most of the uncertain network algorithms cannot be implemented in these systems because of their programming model limitations. Uncertain network algorithms commonly require vertex-wise aggregation (such as finding minimum) over all sample networks, which is not directly supported in conventional graph systems. Moreover, some algorithms perform vertex-wise aggregation multiple times during their execution and their aggregation re-activates a subset of the vertices, which cannot be expressed in conventional systems. Second, even for the algorithms implemented with some modifications to the existing systems, which were limited to SP and PPR, our baseline shows, on average, 2.5× faster performance than the fastest of the three systems. The details of the experimental results are shown in the supplementary material. We now present the evaluation results of Sage-MemOpt and Sage-ExecOpt in comparison to the baseline.

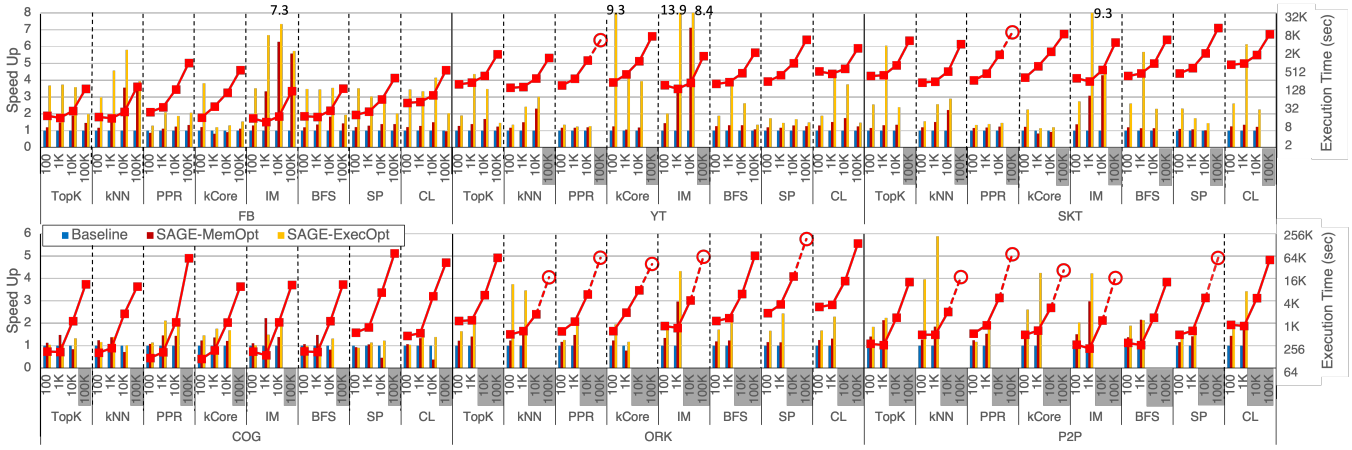


Figure 9: Speedup of Sage-MemOpt and Sage-ExecOpt relative to the baseline (bar plots), and the execution times of Sage-MemOpt in line plots. The o-shaped plots are Sage-MemOpt with reduced cache size to fit in 128GB of memory.

5.3.1 Overall Performance (Memory and Execution). We first demonstrate the amount of memory that is saved by Sage-MemOpt. Figure 8 compares the amount of memory consumed by the baseline and Sage-MemOpt. The gray horizontal dotted line indicates the size of DRAM (128GB) in the machine. For the points that are plotted above this line (128GB), we calculated and plotted the theoretical memory consumption of the baseline and Sage-MemOpt. The baseline runs successfully with 100 and 1K (1,000)[1] sample networks for all the algorithms and datasets. With 10K sample networks, the baseline ran out of memory (OOM) for 33% of the cases (16 out of 48) and with 100K sample networks 75% of the cases (36 out of 48) ran out of memory. In contrast, Sage-MemOpt runs successfully with 100, 1K, and 10K sample networks. With 100K samples, Sage-MemOpt failed for 25% of the cases (12 out of 48). For these OOM cases, we were able to successfully run Sage-MemOpt by reducing the size of the vertex cache so that the entire memory fits into 128GB (with increasing the disk size for a subset of the OOM cases up to 6GB to store the vertex attributes). We report these results when we discuss execution performance. When we reduced the cache size for the baseline system in a similar manner, it failed to run with OOM for all the large networks (COG, ORK, and P2P) with 100K samples because of the memory overhead of the edge existence bitmaps. Overall, excluding the OOM cases, Sage-MemOpt uses only 1.6–63.6% (23.7% on average) of memory compared to the baseline. Moreover, the reduced amount generally increases as the number of sample networks increase. For example, with 100K sample networks, Sage-MemOpt consumes only 13.7% of the baseline's memory usage on average.

Now let us examine the performance of the baseline, Sage-MemOpt, and Sage-ExecOpt. Figure 9 shows the performance of Sage-MemOpt and Sage-ExecOpt relative to that of the baseline in bar graphs and the absolute execution times of Sage-MemOpt in line graphs. The OOM cases for the baseline and Sage-ExecOpt are shaded in dark gray, while the plotted red circles are the 12 cases where Sage-MemOpt ran out of memory with the 100K samples and thus, retrofitted into 128GB by reducing the cache size. We first

[1]For convenience, hereafter, $1K = 1000$

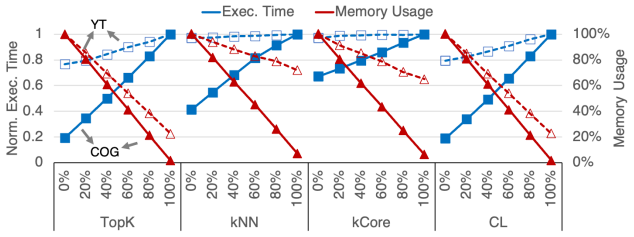


Figure 10: Execution time normalized to full use of deterministic sampling (left y-axis) and memory usage (right y-axis) for running four algorithms on COG with applying deterministic sampling to a subset (0–100%) of edges.

notice that Sage-MemOpt outperforms the baseline by up to 6.8× (IM with YT and 10K samples) and on average by 1.5× even though it uses much less memory. We find that while deterministic sampling trades off memory overhead with computation overhead, the speedup brought by other optimizations exceeds the re-sampling overhead. Moreover, Sage-ExecOpt outperforms the baseline by up to 13.9× (IM with YT and 1K samples) and on average by 2.7×. We also observe that for the COG dataset, the performance gain by Sage-ExecOpt is much smaller than that of other datasets. For COG, Sage-ExecOpt is only 1.3× faster than the baseline on average, and in the worst case (SP with 100 samples), Sage-ExecOpt is 11% slower than the baseline (although with 1K or larger sample sizes Sage-ExecOpt consistently performs better than the baseline). This is because COG has a large number of edges with very low existence probabilities (average 0.18). That makes the overhead of deterministic sampling high for repeatedly generating random numbers for non-existent edges, for which edge computations do not need to run. As such, the performance improvements from deterministic sampling depend on the characteristics of uncertain networks. Thus we next explore how the cost and benefit of deterministic sampling vary for different kinds of uncertain networks.

5.3.2 Deterministic Sampling. Deterministic sampling need not be fully deployed. That is, only part of the edges may be generated using deterministic sampling while the rest are pre-sampled. To

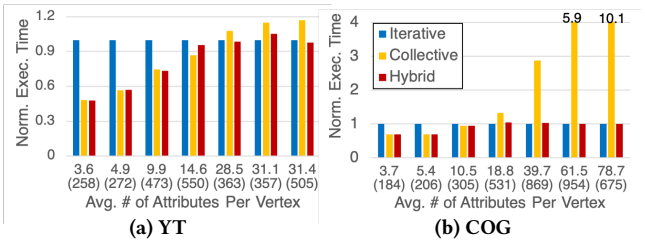


Figure 11: Execution times of SP with seven different edge length setup with iterative, collective, and hybrid gather normalized to that of iterative gather. The x-axis is the average number of attributes per vertex for the seven runs.

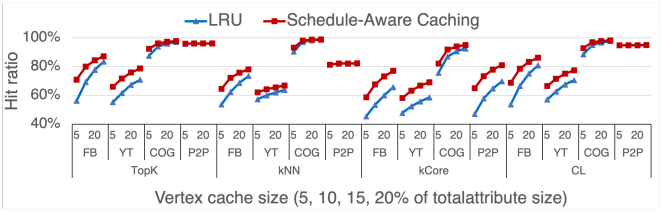


Figure 12: Hit ratio of LRU and our schedule-aware caching.

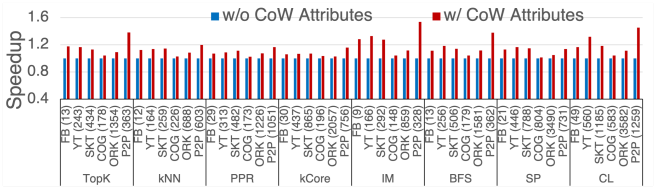


Figure 13: Performance gains by copy-on-write attributes. Execution times without using this feature (w/o CoW Attributes) are shown in parentheses below the x-axis.

investigate the execution time and memory trade-off with partially deploying deterministic sampling, we conduct the following set of experiments. The edges of the vertices are sorted by their in-degrees in ascending order and then the first x% (from 0–100% in 20% increments) of the edges are generated using deterministic sampling, while the rest, i.e. (100–x)% of the edges are pre-sampled.

Figure 10 shows the execution times and the memory usage of the experiments for YT and COG with all SAGE optimizations enabled with cache size of 5% and 10K samples. We compare the results of YT and COG because the results of YT are similar to those of other networks but those of COG are different as previously described. Recall that for COG, because of its low edge existence probabilities, the re-sampling overhead of deterministic sampling is high, resulting in a different memory and computation trade-off.

For both YT and COG, as a larger subset of the edges are applied with deterministic sampling, SAGE uses less memory in all four algorithms. However, the execution times increase at much different rates for the four algorithms and also for YT and COG. Between the two networks, the execution times of COG generally increase at much faster rates than YT because of the high overhead of deterministic sampling in COG. Among the four algorithms, the execution times of TopK and CL increase faster than those of kNN and kCore. This is because the latter are more compute-intensive than the former and thus the overhead of re-sampling is relatively low compared to the execution of the gather operations. For example, consider the experimental results of TopK and kCore for COG. For both the algorithms, deterministic sampling reduces the amount of memory from 81GB to only 1.3GB (62× improvement). While the execution time for TopK increased by 5.3×, the time for kCore increased by only 1.5× because re-sampling is relatively cheap for kCore. These experiments show that for the networks and algorithms where deterministic sampling incurs high computation overhead, we can apply the technique to a subset of edges and control the trade-off between execution time and memory overhead.

5.3.3 Hybrid Gathering. We study the performance impact of hybrid gathering. As its performance is affected by the number of distinct attributes in each vertex, we evaluate hybrid gathering with varying number of per-vertex attributes and compare its performance with iterative gathering and collective gathering. That is, we set up experiments that run shortest paths with an increasing number of distinct edge lengths. The edge lengths of the seven runs are randomly selected to be one of the integer values in ranges 1–1, 1–2, 1–5, 1–10, 1–25, 1–50, and 1–100. We used Zipfian distribution with longer length given higher probability. As the ranges of edge lengths increase, the number of distinct path distances in sample networks also increases making collective gathering less effective.

Figure 11 shows the results of the experiments for YT and COG with 1K sample networks. The x-axis is the average number of attributes per vertex and y-axis is the execution time normalized to that of iterative gather. The average number of vertex attributes ranges between 3.6~31.4 for YT and 3.7~78.7 for COG. Below the x-axis, we show the actual execution times of iterative gather in parenthesis. We can see that when the number of attributes is small (<14.6 for YT and <18.8 for COG) collective gather performs better than iterative gather. For these cases, hybrid gathering executes as fast as collective gathering except for YT with 14.6 attributes, in which case, the repeated attribute type conversion overhead for the source vertex of the gather is relatively large. When the number of vertex attributes is large, iterative gathering is faster. For these cases, hybrid gathering is as fast as iterative gathering, except for YT with 31.1 average attributes, again, due to the conversion overhead.

For all the evaluated algorithms, hybrid gathering performs as fast as the faster of the two methods as in the case study. In particular, for TopK, CL, and BFS collective gathering performs faster than iterative gathering, on average, by 16.4%, 16.0%, and 13.1%, respectively. For these three algorithms, hybrid gathering achieved the same performance as collective gathering as all the vertices maintained the attribute-bitmap tables and no vertex is converted to the attribute array representation. For the other algorithms (kNN, PPR, kCore, IM, and SP), collective gathering runs slower than iterative gathering. For these algorithms, we confirmed that hybrid gathering quickly converts to the attribute array representations for most of the vertices resulting in performance of iterative gathering.

5.3.4 Schedule-Aware Caching. Our caching scheme exploits the execution schedule of edge computations and thus, gives much higher cache performance than the naive LRU policy. Here we quantify the performance gain brought by our caching scheme. To that end we vary the attribute cache size to be 5–20% of the total vertex attribute sizes and run the benchmark algorithms with both LRU and our schedule-aware caching scheme. Figure 12 shows the

Table 3: Execution time and disk usage with large samples

	FB		COG	
Samples	TopK	kNN	TopK	kNN
100K	116s, 8G	147s, 50G	3.1h, 28G	6.9h, 183G
500K	572s, 38G	789s, 251G	15.9h, 140G	37.5h, 920G
1M	1147s, 76G	1703s, 501G	32.6h, 279G	***N/A***, 1.8T
3M	3485s, 228G	***N/A***, 1.5T	99.4h, 836G	***N/A***, 5.5T
5M	5873s, 380G	***N/A***, 2.5T	***N/A***, 1.4T	***N/A***, 8.9T

results of the experiments. It shows that schedule-aware caching results in average and maximum of 6 and 18 percentage points higher hit ratio over LRU (including the results not shown in Figure 12 due to space). This results in performance improving by up to 25% over LRU and by 8% on average (results not shown due to space). Note that the hit ratio of LRU in Sage is already 9 percentage points higher than the baseline due to the data compression effect by hybrid gathering and copy-on-write attributes. Schedule-aware caching further improves the hit ratio by 6 percentage points.

5.3.5 Copy-on-Write Attributes. We investigate the performance impact of copy-on-write attributes. When this optimization is not applied, we allocate and initialize all the attributes before running the first superstep. Copy-on-write attributes reduces the overhead of this initialization at the beginning of analysis. Figure 13 shows the performance of running the benchmark algorithms with and without copy-on-write attributes with 1K samples. It shows that the optimization improves performance by up to 1.53× and on average by 1.15×; the disk space is saved by 21.4% on average. This optimization is most effective for P2P as it consists of several disconnected sub-networks and thus many of the vertices are not reachable when running traversal-based algorithms. The performance for P2P is improved by 1.3× on average using only 6.7% of the disk space.

5.3.6 Scalability of Sage. To understand how Sage scales with the number of sample networks and the uncertain network size, we run two more sets of experiments in much larger scale than the earlier experiments. For these experiments, we only consider TopK and kNN as their memory and disk usage, which is mainly determined by the vertex attribute size, are representative of other algorithms. That is, TopK has a vertex attribute size of 1 byte, as is the case for BFS and CL. The attribute size for kNN is 8 bytes, as are for kCore and IM, which is similar to that of SP (4 bytes) and PPR (16 bytes).

The first experiments evaluate algorithms with large sample networks. To the best of our knowledge, the largest number of sample networks used for uncertain network analysis is 1M (M:million) [47]. Thus, we choose to evaluate even larger sample networks, specifically, up to 5M. For this evaluation, we use the real networks, FB and COG, that provide the uncertainty information in the dataset.

Table 3 shows the results. For FB, both TopK and kNN successfully ran with 100K, 500K, and 1M sample networks. With 3M and 5M sample networks, kNN fails to execute due to the disk space limit. Running kNN with 3M and 5M samples requires 1.5 and 2.5TB of disk space but our machine has only 1TB SSD. To verify that the disk size is the limiting factor, we re-executed kNN for 3M samples with an additional 1TB SSD, that is, with 2TB disk space. In this case, Sage successfully executed the algorithm in 6893 seconds.

For COG, TopK with 100K–3M samples and kNN with 100K and 500K samples successfully executed. Similar to FB, the failing cases are due to the large size of vertex attributes. That is, TopK with 5M samples and kNN with 1M–5M samples failed to run because of the disk space limit as they require 1.4TB–8.9TB disk space that is larger than the one 1TB storage space of our machine. Considering the failed cases of FB and COG, the limiting factor of Sage's scalability for large sample networks is the disk space for storing the vertex attributes of the sample networks.

Table 4: Execution time and disk usage with large networks

	Unif. Prob		Zipf. Prob	
|V|, |E|	TopK	kNN	TopK	kNN
1M, 100M	253s, 1.5G	0.8h, 8.3G	248s, 1.5G	0.6h, 8.3G
5M, 0.5B	1341s, 7.3G	1.4h, 41.5G	1338s, 7.3G	1.1h, 41.5G
10M, 1B	2723s, 14.6G	2.8h, 83.0G	2667s, 14.6G	2.2h, 83.0G
20M, 2B	5718s, 29.3G	5.9h, 166G	5547s, 29.3G	4.6h, 166G

In the second set of experiments, we evaluate the algorithms with much larger uncertain networks than those used in earlier experiments. For these experiments we generate synthetic graphs with up to 20M vertices and 2B (B:billion) edges using the RMAT algorithm [36]. We use the uniform and Zipfian distributions to generate the edge existence probabilities. As previous studies most commonly use 1K sample networks for analysis [26, 32, 47, 57, 63, 87, 89], we evaluate with 1K sample networks as well.

Table 4 shows the results. We observe that Sage runs successfully for up to 20M vertices and 2B edges for both TopK and kNN. Also the execution times for the Zipfian distribution cases are shorter than those of the uniform distribution cases, and more so for kNN. This is because the sample networks with Zipfian distribution have more connected edges and the k neighbors are identified in early iterations. For running kNN for the largest network with 20M vertices and 2B edges, Sage requires 62GB of memory for storing the uncertain network, 46GB for maintaining the vertex cache, and 4GB for storing other data structures such as vertex status bitmaps. The total required memory is 112GB, which is smaller than the 128GB memory of our machine. For any larger network, Sage will run out of memory. Running analysis for such extremely large networks requires partitioning the networks and processing them on a cluster of machines, which we leave as future work.

6 CONCLUSIONS

This paper presented Sage, a system and programming model for uncertain network analysis. Existing algorithms for uncertain networks typically run analysis on sampled network instances and aggregate the analysis results. Sage makes it easy to express these algorithms with its vertex data model and programming interface. In addition, Sage proposes four optimization techniques, namely, deterministic sampling, hybrid gathering, schedule-aware caching, and copy-on-write attributes, which substantially reduce memory usage and improve performance. Through our extensive evaluation with six graphs and eight algorithms, we showed that the four optimizations jointly improve performance by up to 13.9× and on average 2.7× while only consuming, on average, 23.7% of the memory space compared to the baseline. Moreover, all the evaluated algorithms are succinctly expressed in Sage with its high-level programming model tailored for uncertain network analysis.

REFERENCES

[1] Shubhangi Agarwal, Sourav Dutta, and Arnab Bhattacharya. 2020. ChiSeL: Graph similarity search using chi-squared statistics in large probabilistic graphs. *Proceedings of VLDB Endowment* 13, 10 (2020), 1654–1668.

[2] Charu C Aggarwal, Yan Li, Jianyong Wang, and Jing Wang. 2009. Frequent pattern mining with uncertain data. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 29–38.

[3] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[4] Thomas Bernecker, Hans-Peter Kriegel, Matthias Renz, Florian Verhein, and Andreas Zuefle. 2009. Probabilistic frequent itemset mining in uncertain databases. In *Proceedings of ACM SIGKDD International Conference on Knowledge discovery and data mining (KDD)*. 119–128.

[5] Sanjit Biswas and Robert Morris. 2005. ExOR: Opportunistic multi-hop routing for wireless networks. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 133–144.

[6] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1316–1325.

[7] Matteo Ceccarello, Carlo Fantozzi, Andrea Pietracaprina, Geppino Pucci, and Fabio Vandin. 2017. Clustering uncertain graphs. *Proceedings of the VLDB Endowment* 11, 4 (2017), 472–484.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[9] Yifan Chen, Xiang Zhao, Xuemin Lin, Yang Wang, and Deke Guo. 2018. Efficient mining of frequent patterns on uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 2 (2018), 287–300.

[10] John B. Collins and Steven T. Smith. 2014. Network Discovery for uncertain graphs. In *Proceedings of International Conference on Information Fusion (FUSION)*. 1–8.

[11] Graham Cormode and Andrew McGregor. 2008. Approximation algorithms for clustering uncertain data. In *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 191–200.

[12] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *Proceedings of International Symposium on Algorithms and Computation (ISAAC)*. Springer, 403–414.

[13] Soheil Eshghi, Setareh Maghsudi, Valerio Restocchi, Sebastian Stein, and Leandros Tassiulas. 2019. Efficient influence maximization under network uncertainty. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 365–371.

[14] Takayasu Fushimi, Kazumi Saito, Kouzou Ohara, Masahiro Kimura, and Hiroshi Motoda. 2020. Efficient computing of PageRank scores on exact expected transition matrix of large uncertain graph. In *Proceedings of IEEE International Conference on Big Data (Big Data)*. IEEE, 916–923.

[15] Joy Ghosh, Hung Q Ngo, Seokhoon Yoon, and Chunming Qiao. 2007. On a routing problem within probabilistic graphs and its application to intermittently connected networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1721–1729.

[16] GNU. 2018. GNU Time. https://www.gnu.org/software/time/ (visited on 18/09/2022).

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.

[18] Yang Guo, Fatemeh Esfahani, Xiaojian Shao, Venkatesh Srinivasan, Alex Thomo, Li Xing, and Xuekui Zhang. [n.d.]. Integrative COVID-19 biological network inference with probabilistic core decomposition. *Briefings in bioinformatics* 23, 1 ([n. d.]).

[19] Kai Han, Fei Gui, Xiaokui Xiao, Jing Tang, Yuntian He, Zongmai Cao, and He Huang. 2019. Efficient and effective algorithms for clustering uncertain graphs. *Proceedings of VLDB Endowment* 12, 6 (2019), 667–680.

[20] Meng Han, Mingyuan Yan, Jinbao Li, Shouling Ji, and Yingshu Li. 2013. Generating uncertain networks based on historical network snapshots. In *International Computing and Combinatorics Conference*. Springer, 747–758.

[21] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 349–362.

[22] Ming Hua and Jian Pei. 2010. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *Proceedings of the 13th International Conference on Extending Database Technology*. 347–358.

[23] Xin Huang, Wei Lu, and Laks V.S. Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 77–90.

[24] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review* 28, 1 (2013), 75–105.

[25] Ruoming Jin, Lin Liu, and Charu C. Aggarwal. 2011. Discovering Highly Reliable Subgraphs in Uncertain Graphs. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 992–1000.

[26] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. 2011. Distance-constraint reachability computation in uncertain graphs. *Proceedings of VLDB Endowment* 4, 9 (June 2011), 551–562.

[27] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 2 (2017), 305–324.

[28] Vasileios Kassiano, Anastasios Gounaris, Apostolos N Papadopoulos, and Kostas Tsichlas. 2016. Mining uncertain graphs: An overview. In *Proceedings of International Workshop of Algorithmic Aspects of Cloud Computing (ALGOCLOUD)*. 87–116.

[29] Xiangyu Ke, Arijit Khan, and Leroy Lim Hong Quan. 2019. An in-depth comparison of s-t reliability algorithms over uncertain graphs. In *Proceedings of VLDB Endowment*, Vol. 12. 864–876.

[30] Yiping Ke, James Cheng, and Wilfred Ng. 2008. Correlated pattern mining in quantitative databases. *ACM Transactions on Database Systems (TODS)* 33, 3 (2008), 1–45.

[31] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge discovery and data mining*. 137–146.

[32] Arijit Khan, Francesco Bonchi, Aristides Gionis, and Francesco Gullo. 2014. Fast reliability search in uncertain graphs. In *Proceedings of International Conference on Extending Database Technology (EDBT)*. 535–546.

[33] Arijit Khan, Francesco Bonchi, Francesco Gullo, and Andreas Nufer. 2018. Conditional reliability in uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering* 30, 11 (2018), 2078–2092.

[34] Arijit Khan and Lei Chen. 2015. On uncertain graphs modeling and queries. *Proceedings of VLDB Endowment* 8, 12 (Aug. 2015), 2042–2043.

[35] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 169–182.

[36] Farzad Khorasani. [n.d.].

[37] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the International Symposium on High-performance parallel and distributed computing(HPDC)*. 239–252.

[38] Jung Hyun Kim, Mao-Lin Li, K. Selçuk Candan, and Maria Luisa Sapino. 2017. Personalized PageRank in uncertain graphs with mutually exclusive edges. In *Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 525–534.

[39] Kapsu Kim, Myunghui Hong, Kyungyong Chung, and SangYeob Oh. 2015. Estimating unreliable objects and system reliability in P2P networks. *Peer-to-Peer Networking and Applications* 8, 4 (2015), 610–619.

[40] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA.

[41] George Kollios, Michalis Potamias, and Evimaria Terzi. 2013. Clustering Large Probabilistic Graphs. *IEEE Transactions on Knowledge and Data Engineering* 25, 2 (2013), 325–336.

[42] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–46.

[43] Eunjae Lee. 2022. Sage source code repository. https://github.com/mlsys-seo/SAGE/ (visited on 18/09/2022).

[44] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. 2019. Pre-select static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 459–474.

[45] Jure Leskovec. 2009. Stanford Network Analysis Platform. http://snap.stanford.edu (visited on 18/09/2022).

[46] Xiaodong Li, Reynold Cheng, Yixiang Fang, Jiafeng Hu, and Silviu Maniu. 2018. Scalable evaluation of k-NN queries on large uncertain graphs. In *Proceedings of International Conference on Extending Database Technology (EDBT)*.

[47] Yuchen Li, Ju Fan, Dongxiang Zhang, and Kian-Lee Tan. 2017. Discovering your selling points: Personalized social influential tags exploration. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 619–634.

[48] Yongjiang Liang, Tingting Hu, and Peixiang Zhao. 2020. Efficient structural clustering in large uncertain graphs. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 1966–1969. https://doi.org/10.1109/ICDE48307.2020.00215

[49] David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology* 58, 7 (2007), 1019–1031.

[50] Lin Liu, Ruoming Jin, Charu Aggarwal, and Yelong Shen. 2012. Reliable clustering on uncertain graphs. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 459–468.

[51] Wensheng Luo, Xu Zhou, Kenli Li, Yunjun Gao, and Keqin Li. 2021. Efficient Influential Community Search in Large Uncertain Graphs. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[52] Chenhao Ma, Reynold Cheng, Laks V.S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: A motif counting algorithm for uncertain graphs. *Proceedings of VLDB Endowment* 13, 2 (2019), 155–168.

[53] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: Parallel deep neural network computation on large graphs. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 443–458.

[54] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 527–543.

[55] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 135–146.

[56] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 631–643.

[57] Silviu Maniu, Reynold Cheng, and Pierre Senellart. 2017. An indexing framework for queries on probabilistic graphs. *ACM Transactions on Database System* 42, 2, Article 13 (May 2017), 34 pages.

[58] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 1–16.

[59] Kiran Kumar Matam, Hanieh Hashemi, and Murali Annavaram. 2021. Multi-LogVC: Efficient out-of-core graph processing framework for flash storage. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 245–255.

[60] Panos Parchas, Francesco Gullo, Dimitris Papadias, and Franceseco Bonchi. 2014. The pursuit of a good possible world: extracting representative instances of uncertain graphs. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 967–978.

[61] Panos Parchas, Francesco Gullo, Dimitris Papadias, and Francesco Bonchi. 2015. Uncertain graph processing through representative instances. *ACM Transactions on Database Systems (TODS)* 40, 3 (2015), 1–39.

[62] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient probabilistic k-core computation on uncertain graphs. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 1192–1203.

[63] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. 2010. K-Nearest neighbors in uncertain graphs. *Proceedings of VLDB Endowment* 3, 1–2 (Sept. 2010), 997–1008.

[64] Daniel Presser, Frank Siqueira, Luís Rodrigues, and Paolo Romano. 2020. EdgeScaler: Effective elastic scaling for graph stream processing systems. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 39–50.

[65] Ryan Rossi and Nesreen Ahmed. 2012. Network data repository. http://networkrepository.com (visited on 18/09/2022).

[66] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 410–424.

[67] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 472–488.

[68] Arkaprava Saha, Ruben Brokkelkamp, Yllka Velaj, Arijit Khan, and Francesco Bonchi. 2021. Shortest paths and centrality in uncertain networks. *Proceedings of VLDB Endowment* 14, 7 (2021), 1188–1201.

[69] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed SociaLite: A datalog-based language for large-scale graph analysis. *Proceedings of VLDB Endowment* 6, 14 (2013), 1906–1917.

[70] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. 135–146.

[71] STRING. 2000. string functional protein association networks. https://string-db.org/cgi/download (visited on 18/09/2022).

[72] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, Lars J Jensen, and Christian von Mering. 2019. STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research* 47, D1 (2019), D607–D613.

[73] Damian Szklarczyk, Annika L Gable, Katerina C Nastou, David Lyon, Rebecca Kirsch, Sampo Pyysalo, Nadezhda T Doncheva, Marc Legeay, Tao Fang, Peer Bork, Lars J Jensen, and Christian von Mering. 2021. The STRING database in 2021: customizable protein–protein networks, and functional characterization of user-uploaded gene/measurement sets. *Nucleic acids research* 49, D1 (2021), D605–D612.

[74] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 425–440.

[75] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1813–1828.

[76] Dong Wen, Bohua Yang, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2020. Computing k-cores in large uncertain graphs: An index-based optimal approach. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[77] Giscard Wepiw E. and Plamen L. Simeonov. 2006. HiPeer: A highly reliable P2P system. *IEICE Transactions on Infomation and Systems* E89-D, 2 (Feb. 2006), 570–580.

[78] Jun Yan, Lin Zhang, Yupan Tian, Ge Wen, and Jing Hu. 2018. An uncertain graph approach for preserving privacy in social networks based on important nodes. In *Proceedings of International Conference on Networking and Network Applications (NaNA)*. IEEE, 107–111.

[79] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. 2008. Mining significant graph patterns by leap search. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD))*. 433–444.

[80] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-based optimal algorithm for computing k-cores in large uncertain graphs. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 64–75. https://doi.org/10.1109/ICDE.2019.00015

[81] Ye Yuan, Lei Chen, and Guoren Wang. 2010. Efficiently answering probability threshold-based shortest path queries over uncertain graphs. In *Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 155–170.

[82] Heng Zhang, Lingda Li, Donglin Zhuang, Rui Liu, Shuang Song, Dingwen Tao, Yanjun Wu, and Shuaiwen Leon Song. 2021. An efficient uncertain graph processing framework for heterogeneous architectures. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. 477–479.

[83] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. *ACM SIGPLAN Notices* 53, 2 (2018), 608–621.

[84] Bihai Zhao, Jianxin Wang, Min Li, Fang-Xiang Wu, and Yi Pan. 2014. Detecting protein complexes based on uncertain graph model. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 11, 3 (2014), 486–497.

[85] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. 45–58.

[86] Alexander Zhou, Yue Wang, and Lei Chen. 2021. Butterfly counting on uncertain bipartite graphs. *Proceedings of VLDB Endowment* 15, 2 (2021), 211–223.

[87] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2015. Top-k reliability search on uncertain graphs. In *Proceedings of IEEE International Conference on Data Mining (ICDM)*. 659–668.

[88] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316.

[89] Zhaonian Zou, Faming Li, Jianzhong Li, and Yingshu Li. 2017. Scalable processing of massive uncertain graph data: A simultaneous processing approach. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 183–186.

[90] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. IEEE, 649–652.

[91] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Mining frequent subgraph patterns from uncertain graph data. *IEEE Transactions on Knowledge and Data Engineering* 22, 9 (2010), 1203–1218.

I APPENDIX

I.I Gathering Methods

I.I.I Iterative gathering. Algorithm I.1 describes iterative gathering that invokes `gather()` for each sample network if the target vertex of the given edge is triggered in the sample network. It is presumed that the source and target vertices store their attributes in the attribute array representation.

I.I.II Collective Gathering. Algorithm I.2 describes collective gathering. Collective gathering is applied to an edge $s \rightarrow t$, which substitutes the individual invocations of `Gather()` from s to t for all sample networks. In the algorithm, we first allocate proxy source and target vertices (line 3). Then we iterate over the attribute-bitmap table of the s and t vertices (line 2- 3). For the target and source vertex attributes (a_t and a_s), the sample networks having those attribute values are marked in the bitmap B_t and B_s. In line 4 and 5, we check if gather is triggered for any of the sample networks with the attribute pair a_t and a_s, and then run the gather operation with the proxy vertices if necessary (line 6-8). We update the attribute-bitmap table in lines 9, 10 and, 11.

Algorithm I.1: Iterative Gathering

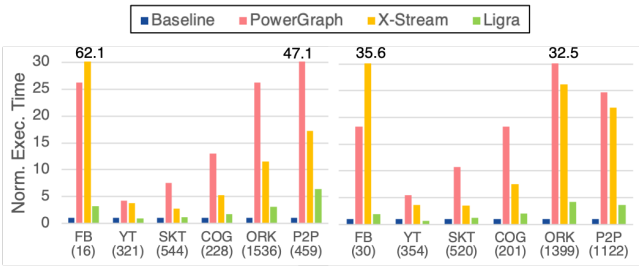
Input: S, T, e, B_e, R,and N. S is source's attribute array, T is target's attribute array, e is the edge connecting the source and target, B_e is e's existence bitmap for sample networks, and R is source's triggering bitmap for sample networks, and N is the number of samples.
Output: T. T is target's updated attribute array

```
1  foreach i in N do
2      if B_e[i] = False or R[i] = False then
3          continue ;
4      v_t = SetVertex(T[i]); v_s = SetVertex(S[i]);
5      v_t.Gather(v_s, e);
6  return T
```

Algorithm I.2: Collective Gathering

Input: S, T, e, B_e, and R. S is source's attribute-bitmap table, T is target's attribute-bitmap table, e is the edge connecting the source and target, B_e is e's existence bitmap for sample networks, and R is source's triggering bitmap for sample networks.
Output: T. T is target's updated attribute-bitmap table.

```
1  AttributeBitmapTable Updates;
2  foreach attribute a_t and bitmap B_t in T do
3      foreach attribute a_s and bitmap B_s in S do
4          Bitmap valid_sample = B_t & B_s & B_e & R;
5          if valid_sample_edge == 0 then
6              continue ;
7          v_t = SetVertex(a_t); v_s = SetVertex(a_s);
8          v_t.Gather(v_s, e);
9          Updates.insert(v_t, valid_sample_edge);
10         T.remove(v_s, valid_sample_edge);
11 T.merge(Updates);
12 return T
```



(a) BFS (b) PPR

Figure I.1: Performance of the baseline system and the existing graph systems for uncertain network analysis. The numbers in parenthesis below the x-axis are the actual execution times of the baseline system in seconds, and y-axis is the execution time normalized to that of the baseline system.

I.II Performance of the Baseline System

This section compares the performance of our baseline system with the existing graph processing systems, that is, PowerGraph [17], X-Stream [67], and Ligra [70] for uncertain network analysis. Because the existing systems are designed for conventional network analysis, we needed to add three modules to support uncertain network analysis. That is, we add the modules for 1) sampling networks, 2) storing the analysis results of sample networks on disk, and 3) loading the analysis results from disk and aggregating them. With the modification, we could implement PPR and BFS with moderate efficiency. The three modules are efficiently implemented; e.g., we used multi-threading for sampling networks and aggregating analysis results. For storing and loading the results, we use the custom binary formats of the graph systems. As X-Stream is a disk-based system, we set its buffer cache large enough to store individual sample network.

Figure I.1 shows the experiental results. We present the results with 1,000 sample networks; generally, we observed that increasing the number of sample networks makes our baseline system to perform better relative to the other systems. As shown in the figure, the baseline runs faster than the other systems for BFS and PPR except for two cases, that is, Ligra for running BFS and PPR with YT; for BFS with YT, Ligra runs 13.3% faster and for PPR with YT, Ligra is 28.8% faster than the baseline. For all other cases, the baseline runs the fastest. On average the baseline is 19.5× faster than PowerGraph, 16.7× faster than X-Stream, and 2.5× faster than Ligra. This evaluation shows that our baseline system achieves reasonably good performance and thus it can be used as the baseline to evaluate the performance of SAGE.