# Bring Orders into Uncertainty: Enabling Efficient Uncertain Graph Processing via Novel Path Sampling on Multi-Accelerator Systems

Heng Zhang*
Institute of Software, CAS
FSA lab, University of Sydney

Lingda Li
Brookhaven National Laboratory

Hang Liu
Stevens Institute of Technology

Donglin Zhuang
FSA lab, University of Sydney

Rui Liu
University of Chicago

Chengying Huan
Tsinghua University

Shuang Song
Meta

Dingwen Tao
Washington State University

Yongchao Liu
Ant Financial

Charles He
Ant Financial

Yanjun Wu
Institution of Software, CAS

Shuaiwen Leon Song†
FSA lab, University of Sydney

## Abstract

Uncertain or probabilistic graphs have been ubiquitously used to represent noisy, incomplete, and inaccurate linked data in many emerging big-data mining and analytics applications. It is impractical to solve uncertain graph problems exactly as it requires to evaluate an exponential number of certain instances (or "*possible worlds*") generated from an uncertain graph. Previously, several CPU-based techniques were proposed to use sampling for uncertain graph processing. However, we observe that (1) they suffer from low computation efficiency and large memory overhead due to unnecessary edge sampling at runtime; (2) they cannot leverage the massive parallelism provided by modern general-purpose accelerators; and (3) there lacks a general programming framework for high-performance uncertain graph processing. To tackle these challenges, we propose a novel runtime path sampling method, which is able to identify and eliminate unnecessary edge sampling via incremental path identification and filtering, resulting in significant reduction in computation and data movement. Centered around this idea, we introduce a general uncertain graph processing framework for multi-GPU systems, named `BPGraph`[1]. `BPGraph` provides general support for users to design and optimize a wide-range of uncertain graph algorithms and applications without concerning about the underlying complexity. Extensive evaluation on a variety of real-world uncertain graph applications demonstrates an average speedup of *26*× (up to *43*×) and better scalability from `BPGraph` over the state-of-the-art frameworks.

## CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**.

*This work is conducted during Heng's visit to FSA lab at University of Sydney.
†Corresponding author.
[1]Beta version can be found at https://github.com/bpgraph/bpgraph


*ICS '22, June 28–30, 2022, Virtual Event, USA*

ACM ISBN 978-1-4503-9281-5/22/06.
https://doi.org/10.1145/3524059.3532379

## Keywords

Uncertain Graph; Sampling; GPU; Performance

**ACM Reference Format:**
Heng Zhang, Lingda Li, Hang Liu, Donglin Zhuang, Rui Liu, Chengying Huan, Shuang Song, Dingwen Tao, Yongchao Liu, Charles He, Yanjun Wu, and Shuaiwen Leon Song. 2022. Bring Orders into Uncertainty: Enabling Efficient Uncertain Graph Processing via Novel Path Sampling on Multi-Accelerator Systems. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3524059.3532379

## 1 Introduction

The use of large-scale graph-structured data has exploded in scientific, data mining and analytics applications in recent years. Some community efforts [14, 28, 38, 45, 53, 54, 59] have been made to efficiently process and analyze these data via exploiting high-performance accelerators under a heterogeneous *scale-up* and *scale-out* setup which has become mainstream node architecture for Top500 supercomputers. Among these important graph analytics, uncertainty is often intrinsic to a wide spectrum of graph applications, which applies to graph data such as noisy measurement in inter-node connection in supercomputing center [38, 55], database querying [7, 12, 25, 26, 29], probability in peer-to-peer network [25], bioinformatics [3, 26, 42], relationship influence in social networks [2, 10, 11], congestion prediction in traffic network [24], etc. In the literature, uncertain graphs (also known as *probabilistic graphs*) have been widely utilized to represent these uncertainties [5, 47]. In an uncertain graph, the existence of connection between two nodes is supposed to be independently indeterminate, and is formulated as a probabilistic edge which is assigned with an uncertainty value. Figure 1 illustrates a sensor network, which encodes the network connectivity probabilities into edge uncertainties. After instantiating the uncertainty of each edge, a set of "possible worlds" are generated to represent all possible instances of the uncertain graph and their probabilities. For instance, for the bottom rightmost possible world in Figure 1(b), which represents the case where all 3 edges exist, its probability is calculated by multiplying the connection probabilities of all the edges, i.e., $0.8 \times 0.5 \times 0.3 = 0.12$. The number of possible worlds equals to $2^{|E|}$, where $|E|$ is the number of edges.

Conventionally, finding the exact solution of an uncertain graph problem requires to iterate through all its possible worlds. As the

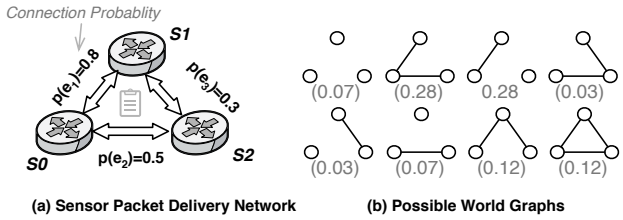



**Figure 1: An example of sensor network with its (a) uncertain graph representation and (b) eight "possible worlds".**

number of possible worlds grows exponentially to the number of edges, it is often unrealistic to get an exact solution. For example, computing the probability of whether there is a path between two vertices on an uncertain graph is *#P-hard* [26, 47]. The rapidly increasing scale of data in these applications has hindered many efforts to seek efficient algorithms for traversing, processing, and mining large-scale uncertain graphs.

**Existing Approaches and Limitations.** Previous works on uncertain graph analysis have sought sampling-based methods to find approximate solutions on uncertain graphs [7, 13, 17, 37, 47]. These methods sample the entire or a part of an uncertain graph to obtain possible world samples. The theory behind these methods is based on the assumption that with a reasonable amount of samples, an approximate solution of the original uncertain graph can be estimated with a certain accuracy guarantee. However, there are three main limitations for applying the existing approaches in HPC environments.

First, *unnecessary edge sampling results in poor computation and memory efficiency*. Our evaluation has shown that the existing approaches suffer from significant performance and memory overhead, especially for large uncertain graphs. Detailed analysis reveals that the main factor behind these overheads is the large amount of unnecessary edges sampled during the execution.

Second, *they lack the support of modern heterogeneous HPC and datacenter architectures which are commonly integrated with one or more accelerators (e.g., GPUs)*. To the best of our knowledge, existing techniques are all built on CPU-based systems. Packed with massive parallelism and high-bandwidth memory, modern GPUs are attractive accelerators for uncertain graph processing [15, 16, 23, 40, 41, 54]. There have been works proposed for deterministic graph processing on multiple GPUs [4, 27, 46, 59]. However, these existing systems cannot handle the probabilistic nature of the uncertain graphs.

Third, *they lack the support of programming API for users to write efficient uncertain graph applications*. Previous uncertain graph processing solutions only provide ad-hoc optimizations on one or a few applications, but fail to provide a general programmable interface for users to effectively implement a wide range of applications. It is unwise to implement different processing strategies and optimizations for different uncertain graph applications which may share many fundamental features. Thus, an efficient, scalable, and programmable uncertain graph processing framework is desirable.

**Approach and Contributions.** To eliminate unnecessary edge sampling and improve computation and memory efficiency, we propose a novel path sampling method. It effectively identifies unnecessary edges prior to sampling, and only considers edges that are on a path between the source and target as useful (Section 3).

Centered around this sampling strategy, we present `BPGraph`, an efficient, scalable, and programmable uncertain graph processing framework that effectively implements path sampling on multi-GPU based HPC systems (Section 4) and scales up heterogeneous node-level computation efficiency. `BPGraph` provides a general programming interface so that users can implement various uncertain graph processing applications with ease. Additionally, it optimizes the data organization and computation patterns to better map uncertain graph processing onto GPUs. To the best of our knowledge, `BPGraph` is the first system design to provide general support for developing and optimizing uncertain graph analytics on multi-GPU based heterogeneous architectures.

Extensive experiments are conducted on eight real-world uncertain graphs to evaluate `BPGraph`, and the results demonstrate that `BPGraph` achieves an up to *43*× speedup (*26*× on average) over the state-of-the-art approaches (i.e., ProbTree [37] and BitEdge-Sampling [69]). `BPGraph` also scales well with the number of GPUs.

## 2 Background & Motivation

### 2.1 Uncertain Graph Basics

First, we give a definition for uncertain graph as follows.

**Definition 1. (Uncertain Graph)** *Let $G = (V, E)$ be a deterministic graph where $V$ is a set of vertices, and $E \subseteq V \times V$ is a set of edges among vertices. An uncertain graph is defined as a triple $\mathbb{G} = (V, E, P)$, where $P$ is a function on edges. For any $e \in E$, $P(e)$ represents the existence probability of $e$. It is obvious that $0 < P(e) \leq 1$.*

We refer $G$ as the corresponding deterministic graph of $\mathbb{G}$. Clearly, $G$ is a special uncertain graph, where $P(e) = 1$ for any $e$. Note that the edge probabilities are independent of each other following by the previous literature. The number of vertices and edges in $\mathbb{G}$ or $G$ can be denoted as the size of vertex list $|V|$ and edge set $|E|$, respectively. It is also worth noting that the probability function $P$ is different from edge weights of deterministic graphs. Without losing generality, we assume all edges have the same weight 1 in this paper. To solve uncertain graph problems, we introduce:

**Definition 2. (Possible World)** *By instantiating an uncertain graph, we denote a possible world $G' = (V, E')$ as a certain instance of the uncertain graph $\mathbb{G}$, i.e., $G' \sqsubseteq \mathbb{G}$. The edge set $E' \subseteq E$ is obtained by executing independent sampling operations on $E$, following the probability function $P$. Thus, each uncertain graph $\mathbb{G}$ yields $2^{|E|}$ possible worlds, based on which edges are selected. Particularly, the possibility of observing a possible world graph $G'$ is calculated by multiplying the probabilities that every edge gets selected or unselected:*

$$\Pr(G') = \prod_{e \in E'} P(e) \prod_{e \in E \setminus E'} (1 - P(e))$$

Many probabilistic problems in data analytic, machine learning, and many other areas employ uncertain graphs to model the inaccurate relationships on datasets of interest. Here, we introduce *reliability* as an application example of uncertain graphs. A wide variety of applications, e.g., network routing [43], network detection [39, 52], route planning [24], web crawling [2, 50], can benefit from a high-performance reliability computation. Given two arbitrary vertices $s$ and $t$ in an uncertain graph, there are four typical variations of $s - t$ reliability:

(1) *Reachability [7].* Compute the probability from $s$ to $t$.

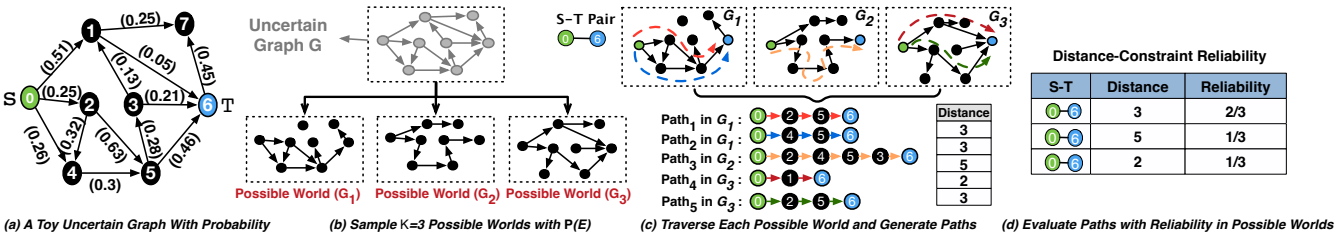



**Figure 2: Execution flow of possible world sampling using the toy graph in (a).**

(2) *Distance-Constraint Reachability [25].* Given a probability threshold $\delta \in (0, 1)$, returns a set of distance $D(s \rightarrow t)$ and the corresponding reachable probability $P(s \rightarrow t)$, with the constraint that $P(s \rightarrow t) \geq \delta$.
(3) *Expected Shortest Distance [63, 66].* On top of the distance-constraint reachability, find the shortest path from $s$ to $t$.
(4) *User-Defined-Constraint Reachability. [26, 70]* On top of the distance-constraint reachability, find eligible paths based on a user-defined constraint. E.g., users may want to filter out paths with too long distance.

In order to compute the exact reliability of an uncertain graph $\mathbb{G}$, we need to iterate through all its possible worlds. Then the reliability of $\mathbb{G}$ can be derived from the reachability between $s$ and $t$ of each individual possible world. Recall that there are $2^{|E|}$ possible worlds in total, making this reliability computation method impractical. Section 2.2 will introduce how previous work uses sampling to find approximate solutions for uncertain graph problems.

## 2.2 Uncertain Graph Sampling

As we state above, it is extremely expensive computationally to get the exact solution of an uncertain graph, which requires to enumerate every possible world. In practice, researchers propose to use sampling methods to get approximate solutions. By solving the problem on randomly selected samples of an uncertain graph, and averaging the solutions on them, an approximate solution of the uncertain graph is obtained. Based on the sampling granularity, existing work can be classified as either 1) *entirety sampling* or 2) *partition sampling*.

**Entirety Sampling.** These methods, e.g., Monte Carlo sampling [48, 69] and recursive sampling [25, 35], randomly sample a certain number of possible worlds from the entire uncertain graph.

Figure 2 illustrates an example of using entirety sampling to solve distance-constraint reachability between $V_0$ and $V_6$. Assume three possible worlds are sampled from the uncertain graph $\mathbb{G}$ based on the independent edge probabilities (Figure 2(b)). After that, we traverse each possible world to figure out all paths from $V_0$ to $V_6$, as shown in Figure 2(c). The distances of these paths are also calculated, which equal to their hop counts. Finally, the distance-constraint reachability between $V_0$ and $V_6$ can be summarized by combining the distances of all paths found. E.g., 2 possible worlds include paths with distance=3 ($G_1$ and $G_2$) out of the total 3 possible worlds, and thus the reliability of distance=3 is calculated as 2/3.

As we will discuss in Section 2.3, entirety sampling results in a lot of redundant sampling overhead due to the similarity between possible worlds, which further causes significant computation and memory overhead.

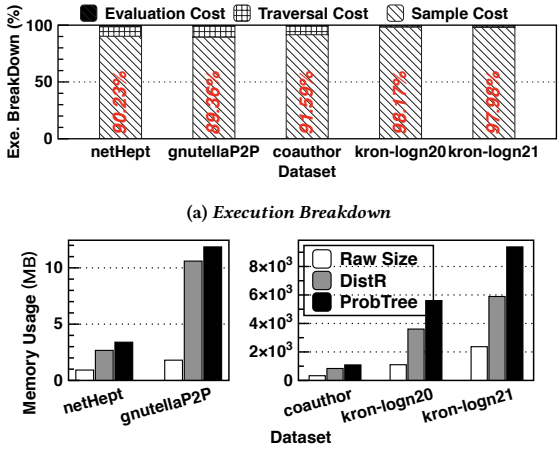



**Figure 3: Performance analysis of uncertain graph processing. We illustrate the results from evaluating a state-of-the-art Monte-Carlo sampling method [69].**

**Partition Sampling.** Instead of sampling the entire uncertain graph to generate possible worlds, partition sampling methods break down the graph into partitions and try to prune useless partitions [8, 37, 62]. These methods organize the mutual dependency between partitions in a tree index structure. Given a reachability query, they find out all partitions that are on the way from the specific source to target vertices, using the tree index structure. Then, a subgraph $G_q$ is created by combining these relevant partitions together, and sampling is performed on $G_q$ instead of the entire uncertain graph. By this way, partition sampling does not sample irrelevant partitions and thus reduces workloads.

The major drawback of these methods is that 1) they still do unnecessary edge sampling because there are useless edges within useful partitions, 2) they require to build the index tree, which is very time consuming, and 3) maintaining the redundant large indexing tree data in memory is difficult and costly, which is unacceptable for GPU platforms which have limited memory capacity.

## 2.3 Challenges & Opportunities

**Unnecessary Edge Sampling.** Figure 3(a) shows the execution time breakdown of an optimized Monte-Carlo sampling method [69], a state-of-the-art entirety sampling based method, running on a 20-core Intel(R) Xeon(R) CPU E5-2698 (512GB memory, detailed configuration in Section 5). The experimental results include the sampling cost (Figure 2(b)), the traversal cost (Figure 2(c)), and the final evaluation cost (Figure 2(d)). It illustrates that more than 90%

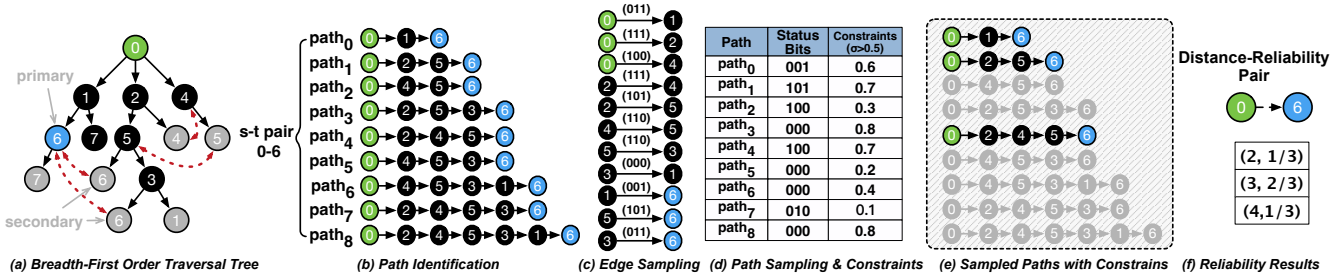



**Figure 4: The workflow of our proposed path sampling.**

time is spent on the sampling. For larger-scale uncertain graphs, the cost of sampling becomes even more significant.

The reason why the sampling phase takes so much time is that existing methods *sample many useless edges* which do not contribute to the results. On one hand, possible world sampling needs to sample every edge of the uncertain graph, no matter whether it is possible for them to be on a path from source to target. For instance, in Figure 2(a), it is obvious to see that none of the paths through edges $V_1 \rightarrow V_7$ and $V_6 \rightarrow V_7$. On the other hand, although partition sampling does not need to sample edges in the pruned partitions, there are still useless edges in the useful partitions. As a result, partition sampling cannot eliminate unnecessary edge sampling within the partitions.

Unnecessary edge sampling has bad consequences: waste of both computation and memory resources because it requires extra memory to cache useless edge sampling results. Figure 3(b) depicts that the memory consumption of two state-of-the-art approaches, DistR [8] that uses entirety sampling and ProbTree [37] that uses partition sampling. The results shows that their memory requirement is more than 5.31× larger than the raw structural data in *kron_g500-logn20* graph, and even 6.57× larger than that in *gnutellaP2P* graph. The partition sampling methods consume more memory because they require extra space to store the index tree. Due to massive caching memory space overhead, existing approaches cannot scale to large-scale uncertain graphs.

**Traversal Redundancy.** Besides, in entirety and partition sampling, there exist a lot of redundant traversals due to the similarity of different possible worlds. For instance, in Figure 2(b), $Path_1$ in $G_1$ and $Path_5$ in $G_3$ are exactly the same. Instead of letting different possible worlds traverse these common paths separately, it will be much more efficient if they are traversed only once.

**Poor Programmability and Generality.** The state-of-the-art methods focus on solving specific uncertain graph problems. For instance, [24] is designed for traffic prediction, while [2] focuses on social network analysis. A general, programmable framework is in need to support the implementation of a wide range of uncertain graph applications.

**Lack of Utilizing State-of-the-art GPU.** Existing uncertain graph processing frameworks are all built upon CPU platforms. Compared to CPU, GPU has shown great potentials for deterministic graph algorithms because of their superior parallel capability [27, 28, 45, 59]. GPU-accelerated data analytic technology has been mainly used for certain graph analysis by now [4, 14, 59], very few literature research on uncertain graph processing. Due to the fact of that SIMD architecture is fit for repetitive computation on regular data, accelerating of uncertain graph processing via GPUs is still challenging, i) how to organize massive possible world in GPU-resident memory via dataset reformation, ii) how to easily express the probability feature of uncertain graph program on parallel SIMD-aware GPUs. This motivates us to leverage GPU's high computation throughput for uncertain graph processing.

## 2.4 Our Goal

To address unnecessary edge sampling and redundant traversals, we aim to identify and filter out useless edges before sampling. Section 3 will introduce *our novel path sampling method* for this purpose. To address the programmability and hardware utilization challenges, we aim to propose a general multi-GPU based uncertain graph processing framework which centers around our path sampling method. Section 4 will discuss the design of our framework.

# 3 Novel Path Sampling

Inspired by our observation that possible worlds share many common paths, we propose a novel *path sampling* strategy. While entirety and partition sampling traverses every possible world after sampling, our path sampling approach requires only a one-time traversal of the uncertain graph before sampling to find all possible paths between the source and target vertices. As a result, the shared paths among different possible worlds are sampled only once which solves the traversal redundancy challenge discussed in Section 2.3. Furthermore, our path sampling only samples edges on possible paths and completely avoids sampling other useless edges. This can address the unnecessary edge sampling challenge in Section 2.3.

## 3.1 Overview

Figure 4 shows how our proposed path sampling works for the uncertain graph in Figure 2(a).

**Path Identification.** First, we traverse the uncertain graph (shown in Figure 4(a)) from the source vertex $V_0$ to find all possible paths that lead to the target vertex $V_6$. To find paths from $V_0$ to $V_6$, we align all vertices along the breadth-first order tree and mark their out edges, then do a bottom-up traversal to find the temporal paths from $V_0$ to $V_6$. After recursively expanding other paths among the inter vertices, all possible paths from $V_0$ to $V_6$ are fetched. For example, after getting the first path $V_0 \rightarrow V_2 \rightarrow V_5 \rightarrow V_6$, we repeatedly add other paths $V_0 \rightarrow V_4 \rightarrow V_5$, $V_0 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5$ from the bottom-up results of $V_0 \rightarrow V_5$ and $V_0 \rightarrow V_2$ until no new vertex is added into the path. Note that, the depth of a path is bounded by the diameter of the graph, which prevents the lengths

of generated paths from getting too large or skewed. Figure 4(b) shows that all paths are found.

**Edge Sampling.** Second, we only sample edges that are on those identified paths. This is the core difference between our path sampling and entirety/partition sampling, which also samples other useless edges. Figure 4(c) shows the sampled edges.

The sampling of different edges are performed independently. A bitmap is used to store the sampling result for each edge. It has $K$ bits ($K = 3$ in Figure 4), and the $i_{th}$ bit represents the result of the $i_{th}$ sampling. If the $i_{th}$ bit $bit_i = 0$, it represents this edge does not exist in the $i_{th}$ possible world $G_i$. Otherwise, $bit_i = 1$ represents it exists in $G_i$. Each edge is sampled $K$ times by generating $K$ random numbers $\{r_1, r_2, ..., r_K\}$ where each of them distributes uniformly in the range of $(0, 1]$. We compare the edge's probability with $r_1, r_2, ..., r_K$. If the probability is larger than $r_n$, we update the $n_{th}$ bit of the bitmap to 1 to mark the edge's existence, otherwise to 0. E.g., the status of edge $V_0 \rightarrow V_1$ is 011 means it exists in the 2nd and 3rd sample but does not in the 1st sample in Figure 4(c).

**Path Sampling.** Third, we combine the edge sampling results to compute the path sampling results. Intuitively, a path only exists when every edge on it exists. Similarly, a bitmap is used for each path to represent its sampling result as shown in Status Bits of Figure 4(d). The path bitmap is calculated by logically ANDing the bitmap of every edge on this path. For example, the bitmap of $path_0$ equals to the bitmap of edge $V_0 \rightarrow V_1$ AND that of $V_1 \rightarrow V_6$, which further equals to $011 \wedge 001 = 001$. Note that the sampling result of each path can be computed independently.

**Constraint based Filtering.** Fourth, candidate paths are filtered based on the path sampling results and user-defined constraints. There is a two-level filter to evaluate the existence of paths. (1) The first filter prunes paths that do not exist in any samples. If all sampling result bits are 0 for a path (Status Bits in Figure 4(d)), it is filtered out. In Figure 4(d), $path_3$, $path_5$, $path_6$, and $path_8$ are pruned by this filter. (2) The second filter prunes paths that do not qualify according to the user-defined constraint criteria. $path_2$ and $path_7$ are pruned by the second filter in Figure 4(d).

**Result Computation.** Finally, we obtain the distance probabilistic results (Figure 4(f)) from leftover paths (Figure 4(e)). This step is same as that of entirety sampling described in Section 2.2.

## 3.2 Computational & Memory Overhead

We model the principle computational and memory overhead of entirety sampling, partition sampling, and our proposed path sampling. Given an uncertain graph $G$, let the number of vertices and edges be $n$ and $m$ respectively, i.e., $n = |V|$ and $m = |E|$, and let the number of possible worlds be $K$.

**Entirety Sampling.** These methods sample each edge of $G$ by $K$ times, and thus the number of total sampling operations is $K \times m$. To memorize the whole set of possible worlds, it consumes $K \times m \times f$ of memory space, where $f$ is the average fraction of edges sampled in all possible worlds and $0 < f < 1$.

**Partition Sampling.** The partition sampling tries to organize the uncertain graph into $P$ partitions, and connect them via a traversal index tree. The s-t query task finds out all partitions along the tree and combine them into one subgraph. The subgraph needs to contain all the traversal paths to give a precise answer for the query. The sampling overhead depends on the edge number of the reduced subgraph $m^p$. The number of sampling equals to $K \times m^p$. Meanwhile, due to the mutual connection between the $P$ partitions, the memory cost of partition sampling will be much larger than entirety sampling, i.e., $K \times m^p \times f + 2P$.

**Our Path Sampling.** Different from sampling the entire graph or a partial subgraph, path sampling achieves the minimal edge sampling number. Its sampling number depends on the number of useful edges $m'$, which equals to $K \times m'$. In the power-law distributed real-world graphs, $m'$ is far less than $m$ in most cases. Similarly, the memory cost of the path sampling is proportional to $K \times m'$. Section 5.2 further evaluates the memory cost on real-world graphs and demonstrates why our method consumes significantly less memory compared to the other methods.

# 4 BPGraph Framework

Building upon our core path sampling method, we propose an efficient GPU-accelerated uncertain graph processing framework, called BPGraph, to provide high-performance, scalability and programmability on multi-GPU systems. In this section, first, we describe the proposed general programming API in BPGraph which allows users to easily define uncertain graph applications (Section 4.1). Then we discuss the implementation aspects of BPGraph, a fast GPU-based design of one-pass path identification and path sampling (Section 4.2), and intra-GPU path sampling optimization, multi-GPU scaling (Section 4.3). Together, they provide an efficient parallel GPU implementation of path sampling.

## 4.1 Path-Sampling Centric Programming

To easily express and debug uncertain graph applications over GPU accelerators, a unified programming API supporting is proposed in BPGraph. Generally, starting from inputting source vertices, uncertain graph applications recursively perform reliability evaluation operations on the set of identified paths until achieving a global reliable result. *User-defined parameters* and *API functions* are provided by BPGraph for user involvements. The parameter option is a simple user involvement which includes the number of samples $K$, accuracy bound, value reliability, etc. User-defined API functions are more expressive which let users to describe the control logic of uncertain graph applications of their interests.

**Application Programming Interfaces (APIs).** The following four API functions are proposed in which stages they are invoked as described in Section 3.1.

- **Path Identification.** We propose a `DispelEntity` function to define activate filter operation of active structural vertices/edges [59], enabling identification of source-to-target paths from graphs.
- **Path Sampling.** We propose a `Initialize` function to initialize the distance of an empty path from identified paths, and define the sampling method utilized. And another function `Expand` is proposed to be repeatedly invoked on all edges of a path to calculate the distance and probability of that path.
- **Filtering & Result Computation.** We propose a function `ReduceVertex` to combine the distance and probability of all available paths on the target vertex.

Listing 1 illustrates the execution flow of BPGraph using these API functions. The input of the execution flow is an uncertain graph, and source and target vertex pairs. The path-centric execution flow is processed under two stages: expand and update the value of paths along the edges (line 5-9), and reduce and calculate the distance and probability values of target vertices (line 12-13).

Taking Figure 4 as an example, given a source vertex 0 and a target vertex 6 in the toy uncertain graph $g$ (Figure 2(a)), the reachability traversal algorithm in Figure 4 aims to answer the distance between them and the reliability value (i.e., the existing probability of paths from 0 to 6). This algorithm is widely used in high performance data-center or sensor network for delivering data packages [51][44].

```
1  void MainProc(Graph g,Vertex srcs[],Vertex tgts[]) {
2      /* Path Identifcation Stage */
3      ConvertGraphToPaths(g, DispelEntity, srcs, tgts);
4      /* Path Sampling Stage */
5      parallel-for Path p in g.getPaths(srcs, tgts)
6          Initialize(p);
7          // Nested parallel processing edge along paths
8          parallel-for Edge e in p.getEdges()
9              Expand(p, e);
10         synchronize; //Synchronize cooperative threads
11     /* Result Computation:Reduce from all paths */
12     parallel-for Vertex (s, v) in (srcs, tgts)
13         ReduceVertex(g.getPaths(s, v), v);
14     synchronize; //Synchronize cooperative threads
15 }
```

**Listing 1: Pseudo code for path-based execution flow.**

```
1  /* Path Intial Identify Phase */
2  extern void DispelEntity(){
3      DistanceConstrain=5;ReliabilityThreashold=0.1;}
4  /* Path Sampling Phase */
5  __device__ void Initialize(Path p) {
6      p.distance = 0; }
7  __device__ void Expand(Path p, Edge e) {
8      p.distance += e.distance;
9      p.prob = OP_AND(p.prob,e.prob); }
10 /* Result Computation Phase */
11 __device__ void ReduceVertex(Path pArray[], Vertex v) {
12     for path p in pArray {
13         if((p.distance < DistanceConstrain)
14             && (p.prob > ReliabilityThreashold))
15             // Use expected-reliable formulation
16             atomicAdd(v.distance,
17                 OP_MUL(p.prob, p.distance));
18     }}
```

**Listing 2: Source-to-target query implementation in BPGraph.**

Listing 2 exhibits how to implement the reachability traversal algorithm in BPGraph. First, before execution, the following global constraints are defined through parameter-based user involvement, i.e., *ReliabilityThreshold=0.5* and *DistanceConstrain=5* to filter out unqualified paths. Subsequently, as shown in Listing 2, we first define the initial distance of an empty path to be 0 (line 3). In this example, the distance of a path is defined as the summation of all edges' distances (line 7), e.g., the distance of path [0 1 6] is calculated as 2 in Figure 4(d). Besides, a path would exist in a possible world only if all its edges exist in that possible world (line 8), which is exploited to filter the status bits in Figure 4(d). Finally, by aggregating the results of corresponding paths, *ReduceVertex* is used to calculate the distances and probabilities of vertices via multiplying distance and probability for an expected-reliable result, in which

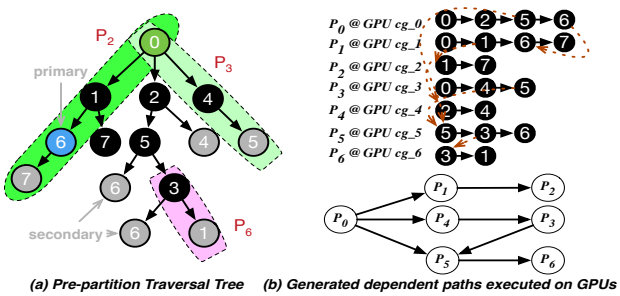



**Figure 5: Dependency generation in path identification.**

the paths need to satisfy the global probability and distance constraints (line 12-17). The final expected-reliable distance of vertex 6 for reachability traversal from 0 is $2 * \frac{1}{3} + 3 * \frac{2}{3} + 4 * \frac{1}{3} = 4$ (Figure 4(f)).

**Generality of our APIs.** To demonstrate the expressiveness of our APIs, in addition to the aforementioned source-to-target query application, we have further developed a number of other uncertain graph applications with our APIs, which include source-target query (a.k.a, s-t query), k-nearest neighbor search, breadth-first search, any-pair shortest paths. These algorithms are all built upon the path sampling model.

## 4.2 GPU-based Design for Asynchronous Path Identification and Sampling

As we mentioned in Section 3.2, path-sampling methodology achieves the minimal sampling overhead and consumes significantly less memory compared to other methods. Building a parallel uncertain graph processing framework over GPUs is still challenging [33, 34, 56]. This subsection focuses on the implementation of challenging phases, which are parallel path identification (Figure 4(a)) and sampling (Figure 4(b)). To enable fast path identification, inspired by iterative graph processing methods [32, 37, 64], BPGraph maintains the entire uncertain graph under a structure of breadth-first ordered tree (Figure 5) to help optimize graph locality.

**Asynchronous Path Identify via Dependency.** As we can see from the breadth-first ordered tree, a large number of paths of varying lengths will result in space explosion. Before identifying massive paths, BPGraph introduces two steps to pre-partition paths via a dependency tree building technique: (a) *breath-first layer-aware decomposition:* we first perform a partition stage to divide the ordered tree into linked paths (as shown in Figure 5); (b) *construct dependency tree of linking paths:* connecting the corresponding in-neighbors of the start vertex to create dependency between the linked paths. For example, according to the dependency relationship of the start vertices 0, 2, 3, both of $P_1$, $P_4$ and $P_5$ are dependent on $P_0$. Furthermore, these bridging vertices are marked with two flags *primary* and *secondary* (e.g., marked vertex 6 in Figure 5(a)), which are used for next inter-path state synchronization.

Following the generation of the dependency tree, BPGraph drives GPU threads to execute asynchronously over consecutive accessing of edges along paths. In particular, when executing source-to-target query requests, each path is asynchronously dispatched to GPUs as a single processing workload unit. GPU thread or cooperative group sequentially checks the corresponding paths, and synchronizes

along the dependency (Figure 5(b)). For example, during processing s-t pair (0-6), GPU threads propagate the starting vertex along paths, and identify 0->2->5->6 and 0->1->6 during first round. Under the following round, other parts of path among 0-6 are identified after synchronizing primary and secondary vertices, e.g., paths 0->4->5->3->6 are along $P3$ and $P6$ during 2nd round , and path 0->2->4->5->3->6 are along $P0$,$P4$ and $P5$ during 3rd round. Note that, same as previous uncertain graph processing work [8, 25, 37, 47, 69], these simple paths with length shorter than the diameter of the graph ensure no vertex that appears more than once in the sequence, naturally eliminating circles along the paths.

In addition, we introduce a consecutive formation as three arrays for caching identified paths: 1) the increasing number of paths of varying lengths; 2) each path's first edge offset; 3) edges storing along paths, with the source and destination of edges represented by two consecutive vertices; e.g., 4 edges of path 0->2->5->6 is stored as [0 2 5 6].

**Incremental Path Sampling.** Figure 4 has shown the overall path sampling process in BPGraph. From path identification (b) to edge sampling (c) in Figure 4, different paths share common edges, for example, edges (0,2), (2,4), and (4,5) are in both $path_7$ and $path_8$. If each path is sampled independently, it will result in duplicate sampling. A straightforward method is to group the edges of previously identified paths into a new edge list and sample these edges in parallel. However, re-organizing identified paths into edges brings extra random accessing overhead and necessitates pre-allocating large cache space for the *edges*.

Instead, we design an incremental hierarchical path sampling strategy to reap the benefits of GPU cooperative group programming. Other than sampling every path or edge, this strategy eliminates many redundant sampling workloads from early edge filters (e.g., edge (3,1)). Cooperative group (cg) allows kernels to dynamically organize groups of threads, ensuring synchronize groups of threads smaller than thread blocks, and software reuse in the form of "collective" group-wide function interfaces [21]. The core idea of our incremental path sampling is to sample incrementally and judge whether to add the sampled equilong sub-paths to paths. The lengths of these sub-paths correspond to the GPU multi-threading resources available at runtime, i.e., cg tiles. In particular, to store the common sub-paths, a shared cached array is kept in global memory to mark whether or not the sub-paths have been sampled. Each thread block is partitioned into multiple "tiles" via *cooperative group* function *tiled_partition()*, in which the template parameter of this function is determined by the lengths of sub-paths. One path sampling workload is assigned to one group. If the sub-paths are not sampled, threads in each tile sample edges based on their probability and adds their partial status bits to other threads by finding *shfl()* operation. The rank thread 0 sums up the distance and reliability value of this path. Before cooperative group synchronization (*sync()*), BPGraph generates the existence bits of each path one by one by merging and prefix sum over edges. After filtering the available paths, by executing user-defined kernel functions in parallel (Listing 1), the reliability value and weights of vertex from source vertex to the target vertex of paths are updated asynchronously.

To this end, the application results and their reliability value are achieved via a selective method to aggregate values along different length of distance paths.

(1) **Thread-Level Path Sampling.** When the length of paths are in small size (<32), the sub-warp kernel processes several identified paths in a single warp and requires fewer threads (32, 16, 4) with *tiled_partition<num_thread>*.
(2) **Warp-Level Path Sampling.** When the length of paths are in medium size (<1024), sampling kernel requires less than the maximum thread block size (1,024) with *coalesced_threads*.
(3) **Grid-Level Path Sampling.** When the length of paths are in large size (> 1024), the grid kernel processes paths in several thread blocks and sampling requires more than 1,024 threads with *thread_block* and even *grid_group* on devices.

## 4.3 Scalable BPGraph Implementation on Multiple GPUs

With the size of uncertain graph increasing, scaling BPGraph to multi-GPU systems (shown in Figure 6(a)) will become more desirable and beneficial. BPGraph tackles the scalability challenge for processing large-scale uncertain graph via a streaming graph partition and workload migration design. To fully utilize the aggregated GPU memory, BPGraph distributes the entire uncertain graph into the multi-GPU memories. Figure 6 illustrates the essential communication and synchronization requirements of the multi-GPU version of BPGraph.

**Streaming Uncertain Graph Partition and Allocation (❶ in Figure 6).** There are many well-known certain graph partitioning approaches, such as vertex partition in GraphX[61] and GraphLab[19], Metis [31] and grid partition in GridGraph [68]. In BPGraph, we uses *edge partitioning* method [27] to achieve the fast and flexible uncertain graph partition.

Assuming there are $D$ devices, all edges are partitioned into $\#GPU$ disjoint partitions $P_i (1 \leq i \leq D)$, where $E = \biguplus_{i=1}^{D} P_i$. Using the edge partitioning approach, *CSR*-formatted inputs can be efficiently partitioned via fast scanning the row indices of graphs. In our experiments, partitioning million-edge graph only takes 1.5-4.2 milliseconds. The corresponding partition of distance and edge probability value are cached in each GPU, which allows each GPU synchronizes their own copy of vertex array using *GPUDirect PeerToPeer* communication by only updating their own potion of results. By exploiting the optimized CUDA I/O primitives (*cudaMemAdvise* and *cudaMemPrefetchAsync*), loading of uncertain graph partitions in streams benefits from efficient data prefeching and a significant reduction in page fault before kernel launching. Further, following the previous design, the primitives of cooperative group enable global synchronization patterns across multiple GPUs within CUDA . We design multi-device cooperative group for multi-accelerator management via the initialization *cudaLaunchCooperativeKernelMultiDevice* and enabling synchronization of thread groups.

**Dynamic Workload Migration (❷ in Figure 6).** Due to the irregularity and probabilistic nature of uncertain graphs, GPU-accelerated systems have a difficult time balancing workloads: 1) the path identification workload will require dynamically filtering redundant unavailable vertices and edges; 2) the path sampling workload cannot be balanced across different devices due to path length differences. These types of workloads distributed across devices and SMXs may result in side loading and some devices being

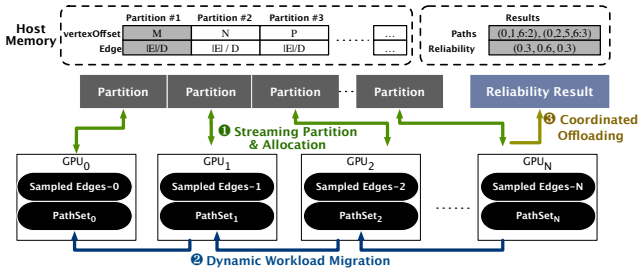



**Figure 6: Data management and Inter-GPU communication over multi-GPU systems.**

released earlier than others at runtime. To address these scalability issues, we develop a dynamic workload migration strategy.

During the path identification stage (Figure 4(a)), we design a dual buffer to handle the workload balancing, inspired by only transferring active workload strategy [27]. In particular, each thread block of GPU constructs two dynamic-sized buffers, i.e., *in-buffer* and *out-buffer*, which is utilized for synchronization and migration. For example, if we need to traverse other edges stored in other devices and migrate the redundant edges of paths from GPU-1 to GPU-D, the buffer of GPU-1 will fill into the path's current destination vertex id after determining whether any of the warp lanes has a responding workload to transfer. Through ring topological synchronization, the buffers of devices are synchronized and the updated messages are sent along GPU-1, GPU-2,..., GPU-D.

During the path sampling stage (Figure 4(c)), the host CPU estimates the sampling workload for each GPU by calculating the number of edges that it needs to sample ($|E|_{sample}$). Then, BPGraph balances the sampling workload by moving sampling edges from overloaded GPUs to under-utilized ones.

**Coordinated Data Offloading (❸ in Figure 6).** Finally, to maintain the identified paths and reliability results, we construct a shared zero-copy buffer to store a separate path array for each GPU after synchronizing sampled edge set. The path evaluations are independent with each other. Thus, each GPU samples edges and filters paths asynchronously during the sample runtime. BPGraph chooses shared zero-copy buffers to hold the path, sampled edges, and the evaluated vertex values (distance and reliability values). During runtime, GPU threads directly update the $Edge_{id}$ in paths and filter the pertained part of edges. At the end, all evaluation results of vertices are collected from individual GPU and written to the CPU buffer.

## 5 Evaluation

**Platform.** Table 2 shows detailed platform configuration. We perform our GPU evaluation on GPU evaluations an NVIDIA DGX server with 8 NVIDIA V100 GPUs. The host system of DGX server consists of two 20-core Intel(R) Xeon(R) CPU E5-2698 v4, and 512GB DDR4 main memory, running with Ubuntu 18.04 (kernel 4.15.0) and CUDA 11.0.

**Datasets.** Table 1 shows the real-world graphs with a broad range of sizes and features from Stanford Large Network Dataset Collection[2], Network Data Repository[3], etc. The soc-twitter and com-friendster [1] are collected from the real-world social network.

[2]http://snap.stanford.edu/data/
[3]http://networkrepository.com/index.php

**Table 1: Real-world graph datasets used in this paper. $S_G$ represents the raw graph size with probabilistic edge list format.**

| *Dataset* | *Vertices* $\|V\|$ | *Edges* $\|E\|$ | *Size* $S_G$ | *Prob* $\Pr(E)$ | *Avg. Degree* $\overline{D}$ |
|---|---|---|---|---|---|
| *netHept*[1] | 15,233 | 62,774 | 921KB | 0.04±0.04 | 4 |
| *gnutellaP2P*[1] | 62,586 | 147,892 | 1.8MB | 0.23±0.20 | 2 |
| *coauthor-DBLP*[50] | 540,486 | 15,245,729 | 331MB | 0.11±0.09 | 28 |
| *kron-logn21*[50] | 1,544,088 | 91,042,012 | 2.3GB | 0.33±0.28 | 58 |
| *soc-twitter*[30] | 28,504,110 | 531,000,244 | 14GB | 0.46±0.28 | 18 |
| *uk-2005*[1] | 39,454,748 | 936,364,284 | 20GB | 0.32±0.25 | 24 |
| *com-friendster*[1] | 65,608,366 | 1,806,067,135 | 41GB | 0.52±0.25 | 29 |

**Table 2: Platform Specification.**

| **NVIDIA DGX Server (8 x GPU V100)** | |
|---|---|
| Shading Unit | 5120 @ 1530MHz |
| On-chip Storage | L1 Cache/Shared: 48KB x 80 |
| | L2 Cache: 6MB |
| Default Memory | HBM, 32GB, 320GB/s |

We use 7 publicly available real-world graphs. The edge probabilities in the first 2 datasets come from real-world applications, using the same configuration in [37, 62], while the probabilities in the latter 5 datasets are randomly assigned within the specified value range.

**Parameter Setting.** To generate fair s-t query pairs, we select 10 different source vertices, uniformly at random from the datasets. Next, the target vertices are chosen from $n$ hops from the source vertices, uniformly at random, in which $n$ is randomly selected between 2 and the graph diameter. The reported results of s-t query are calculated by averaging those of all pairs. Initially, the value $K$, i.e., # samples, is 100. It increases at a step of 200 till the results converge.

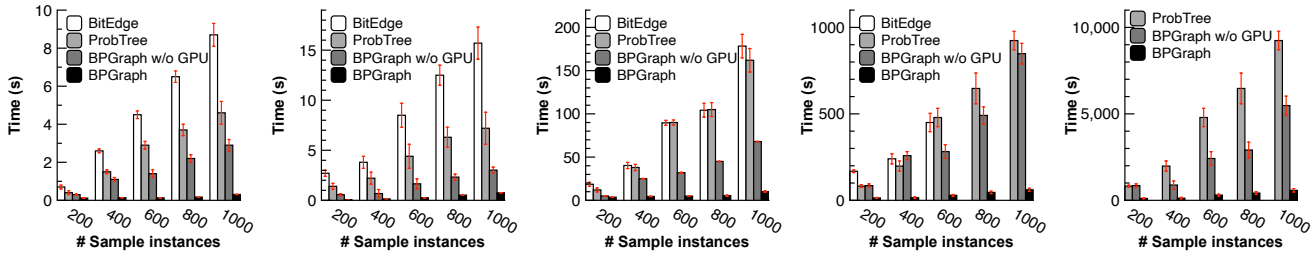
**Benchmarks.** We adopt the three benchmarks, i.e., source-to-target query, k-nearest neighbors, and any-pair shortest path, which are popular benchmarks in previous uncertain graph studies. Given an uncertain graph $\mathbb{G}(V, E, P)$, a possible world $G \sqsubseteq \mathbb{G}$, and a distance function $d$, we give the formulas of these problems as the following.

*(1) Source-to-target query (ST)* is defined as returning the distance with probability greater than a reliability threshold between the given two vertices $s$ and $t$. ST query aims to compute the distance between $s$ and $t$ based on the distance function.

*(2) K-nearest neighbors (KNN)* is defined as returning top-$k$ vertices with minimum distances and reliable probability from the given vertex $s$. Given a node $s(s \in V)$, an integer $k > 0$ and a reliability threshold $\sigma$, the $k$-nearest neighbors query (k-NN) aims to find a set of nodes $C$ such that for any $r \in C$, the value of $d(s, r)$ (marked as $\mathbb{D}$) is in the top-$k$ list w.r.t. the function $d$ and their probability $p_{d(s,r)}(\mathbb{D}) = \sum_{G|d_G(s,r)=\mathbb{D}} Pr(G) > \sigma$ [49].

*(3) Any-pair shortest path (APSP)* is defined as returning paths with the minimum distance and reliable probability. Given any-pair nodes $S$ and $T$ (S,T⊂V) and a reliability threshold $\sigma$, it aims to compute the set of the shortest distance paths between $\forall s \in S$ and $\forall t \in T$ based on distance function $d$. Meanwhile, the probability of each path need to fit the reliability threshold, i.e., $p_{path(s,t)}(\mathbb{D}) > \sigma$.

For the KNN problem, BPGraph firstly generates multiple sets of neighbors of node $s$ and then evaluates them with reliability and shortest distance until finding a full set of top-$k$ neighbors. The APSP problem in the uncertain graphs is different from certain




(a) *netHept* (b) *gnutellaP2P* (c) *coauthor-DBLP* (d) *kron-logn21* (e) *soc-twitter*

**Figure 7: *GPU-accelerated performance comparison. Lower is better.***

ones. BPGraph counts all input source-to-target pairs. User-defined constraint criteria is utilized to filter active paths, e.g., in Figure 4(d), the third column values of paths need to be greater than 0.5. User-defined filtering criteria is configured as a distance constraint to limit the number of edges along paths.

**State-of-the-arts.** We evaluate our system by comparing with three state-of-the-art methods, and report the number of samples required for convergence, running time, and memory usage for all systems. Our datasets and source code will be publicly available. The four state-of-the-art algorithms are as following Table 3. For comprehensive and more fair comparisons, we also implement GPU-accelerated MC-Sampling, BitEdge Sampling method, in which we generate and traverse the $K$ possible worlds resided in the GPU memory. Meanwhile, the state-of-the-art CPU-based algorithm BitEdge, ProbTree, and DistR is enhanced using OpenMP to leverage multi-core CPUs on our evaluation platform which are able to execute 40 threads simultaneously.

**Table 3: Methods in Evaluation.**

| Abbr. | Framework |
|---|---|
| MonteCarlo (MC) | Monte-Carlo sampling regarding to uncertain graph processing, we report the basic entirety sampling based on this sampling method [47]. |
| BitEdge (BE) | Simultaneous sampling method simultaneously processing massive possible worlds with compact bit-aware marked edges [69]. |
| ProbTree (PT) | ProbTree sampling method by partitioning graphs and generates a small uncertain subgraph for querying purposes [37]. |
| DistR (DR) | Distributed reliability-aware uncertain graph sampling method (DistR) based on partition sampling [8]. |

## 5.1 Comparison with State-of-the-art

Table 4 illustrates the performance of BPGraph and state-of-the-arts. We compare their performance for three applications: s-t reliability query, k-nearest neighbor (KNN), and any-pair shortest path. BPGraph runs on a single GPU in these experiments while others run on multi-core CPUs.

As Table 4 shows, BPGraph significantly outperforms other methods in all applications. For the s-t query, BPGraph is on average 39× and 30× faster than entirety sampling methods, i.e., MC sampling and BitEdge Sampling, respectively. Compared to the partition sampling method (ProbTree), BPGraph is 26× faster. For KNN and shortest path, BPGraph presents even higher speedups, e.g., it achieves an average speedup of 69× and 43× compared to MC sampling and ProbTree respectively. It is because these two applications have much larger workloads due to recursive traversal and sampling. For the reason of BPGraph's superior performance, it is because 1) BPGraph only traverses the graph once and also samples useful edges only, 2) the high-parallelism computing capability of GPU over multi-core CPUs, and 3) the high memory bandwidth of the NVIDIA V100 GPU over the CPU.

To further compare to *ProbTree*, we evaluate its index building overhead, and breakdown the execution time of ProbTree. The evaluation of ProbTree on netHept and gnutellaP2P shows that the index building takes 79% and 84% of the total execution time, respectively. Since ProbTree partitions raw graphs into fully-connected cliques, the index building overhead comes from both partitioning and the reliability information re-computation, which is significantly reduced by the simple path identification in BPGraph.

Moreover, to show the benefits of path sampling, we implement a version of CPU-based BPGraph without utilizing GPUs. Figure 7 compares the performance of BPGraph on GPU with one source-target pair query, BPGraph on CPU, BitEdge (BA) and ProbTree (PT), which performs best among three state-of-the-art methods. Even without GPU acceleration, BPGraph on CPU still achieves better performance than ProbTree, which indicates the effectiveness and efficiency of our proposed path sampling method. Also, we can see that the performance of BPGraph on GPU achieves over 6.15-23.5× improvement compared with BPGraph on CPU.

## 5.2 Evaluation on Memory Cost

Figure 8 studies the memory consumption in BPGraph, which has the lowest memory overhead compared with other methods. For instance, BPGraph consumes 16GB memory space on twitter graph, which is 32% of the memory overhead of ProbTree. This is because BPGraph only stores useful edges and paths. Compared to the entirety and partition methods, BPGraph does not need to cache all the possible paths, and we only store the traversal paths in the breadth first ordered tree via an indexing way, and format each path as a consecutive array storing the successive vertex IDs, which significantly reduces the caching space. For kron-logn21, path caching consisting of the structural and probabilistic data totally consumes 4.2GB memory (13.12% resident memory). For largest graphs uk-2005, it requires 30.4GB of GPU memory in BPGraph and therefore

Table 4: End-to-end performance comparison between BPGraph and the state-of-the-art approaches. We report the execution time on 100 samples (K=100). (We report the time in seconds and mark the speedup of BPGraph over best algorithms in parentheses, and the index building time of PT is also considered for fair comparison of end-to-end performance.).)

| Graph | source-target query | | | | k-nearest neighbors | | | | any-pair shortest path | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MC | BA | PT | BPGraph | MC | BA | PT | BPGraph | MC | BA | PT | BPGraph |
| *netHept* | 24.3 | 25.1 | 13.2 | 2.4 (5.5) | 49.8 | 45.2 | 6.4 | 2.8 (2.3) | 35.1 | 16.8 | 13.1 | 1.2 (15.5) |
| *gnutellaP2P* | 38.2 | 23.9 | 11.4 | 4.3 (2.6) | 117.5 | 74.3 | 35.9 | 6.1 (5.9) | 45.9 | 19.6 | 26.4 | 2.7 (7.2) |
| *coauthor* | 368.6 | 268.2 | 289.4 | 10.8 (24.8) | 1793.0 | 1288.6 | 1031.2 | 43.1 (23.9) | 286.2 | 238.0 | 286.0 | 4.3 (55.3) |
| *kron-logn21* | 2471.1 | 1085.4 | 1635.7 | 30.6 (35.5) | 3303.2 | 2372.9 | 5293.5 | 138.0 (17.2) | 2580.5 | 1294.3 | 1229.4 | 20.8 (59.0) |
| *soc-twitter* | 38816.2 | 12835.5 | 9494.2 | 486.3 (19.5) | 43291.6 | 17416.9 | 6486.9 | 1245.2 (5.2) | - | 13985.4 | 9620.4 | 325.1 (29.6) |
| *uk-2005* | - | - | 21081.0 | 607.6 (34.7) | - | - | 12113.8 | 1501.2 (8.51) | - | - | 23940.2 | 894.2 (26.7) |

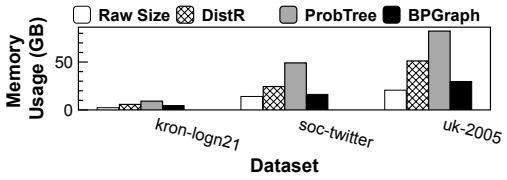



Figure 8: Memory cost comparison between DR, PT and BPGraph.

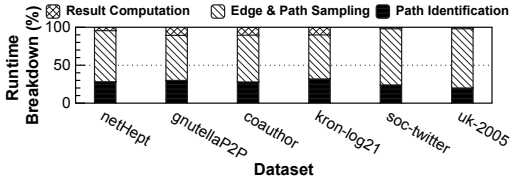



Figure 9: Execution breakdown of BPGraph over three stages of path sampling model (path identification, edge and path sampling, result computation).

fits into V100, while other approaches fail to fit. These results also verify our theoretical memory consumption analysis in Section 3.2.

## 5.3 Execution Time Breakdown

Further, we evaluate the computation time breakdown, as shown in Figure 9. Compared with the breakdown of MC-Sampling shown in Figure 3, we observe that the sampling phase takes much less portion of the entire execution in BPGraph. Since we eliminate all unnecessary edge sampling, which reduces the sampling time, the portions of path identification and result computation phases increase in the overall time.

The path identification phase overhead is a key component of the overall execution time, which generates the whole path set of given vertices, as shown in Figure 4(b). Table 5 illustrates its overhead with various number of source-target pairs for four graphs. From the reports of the uncertain graph *soc-twitter*, BPGraph executes the traverse of path identification phase in 8.38-85.20s for the 10-300 source-target queries. The path identification takes 20.3-31.9% of the total execution time.

## 5.4 Evaluation on Accuracy

For all sampling based uncertain graph processing approaches, the solution accuracy is apparently affected by the number of samples. We aim to study the relationship between the number of samples $K$ and accuracy here, using the s-t query application on the twitter graph as an example.

We do the accuracy study for BPGraph, BitEdge-Sampling, and ProbTree. We define the accuracy error as the difference percentage between the approximate solution of sampling methods and the exact solution of the uncertain graph. The accuracy error of reliability is reported with respect to MC sampling, which is computed as: $AccuracyError(K) = \frac{1}{100}\sum_{i=1}^{100}\frac{|R(s_i,t_i,K)-R_{exact}(s_i,t_i)|}{R_{exact}(s_i,t_i)}$. $R_{exact}(s_i,t_i)$ denotes the reliability achieved from $s_i$ to $t_i$ using the exact solution, and $R(s_i,t_i,K)$ denotes the reliability estimated with $K$ samples.

We evaluate the accuracy error values for $K$ from 100 to 1000, with a step of 100. The three methods have fairly low errors at $K = 1000$. For BitEdge and ProbTree, they accuracy error improves from 10.6% to 1.5% and 15.73% to 0.97% when $K$ changes from 100 to 1000. On the other hand, the accuracy error of BPGraph changes from 4.35% to 0.21%. Clearly, BPGraph achieves better accuracy with the same number of samples. The high accuracy of path sampling model comes from that the model only cares about the dependency between vertices on the paths and directly applying the reliabilities to the target.

Note that Table 4 compares the performance of different approaches under the same sample number. We have shown that BPGraph needs less samples to achieve the same accuracy compared with other methods. As a result, BPGraph will have even better performance if the goal is to let all approaches converge to the same accuracy.

Table 5: Performance of generating traversal paths.

| Number of ST Pairs | Datasets | | | |
|---|---|---|---|---|
| \|Q\| | netHept | coauthor | kron-logn21 | twitter |
| 10 | 0.27s | 0.31s | 1.38s | 8.38s |
| 50 | 0.85s | 1.23s | 3.62s | 26.20s |
| 100 | 1.58s | 3.26s | 11.6s | 34.87s |
| 300 | 3.95s | 12.42s | 22.8s | 85.20s |

## 5.5 Evaluation on Impact of Distance Metrics

Further, to figure out the impact of distance metrics (illustrated in Section 2.1), we exploit the 100-sample ST query application over three typical variations of reliability definition, i.e., *reachability*, *distance-constraint reachability*, and *expected shortest distance*. These three metrics are commonly exploited in uncertain graph literature for distance computation [26][51][25][49]. From the results in Figure 10, we can see that using three metrics, the overall running time does not change significantly (<4.6% performance variation). The reason for this is that the distance calculation consumes less

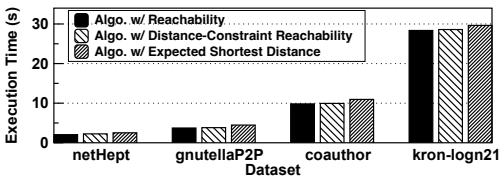



**Figure 10: Runtime of BPGraph over three different variations of s-t reliability measurements (illustrated in Section 2.1). Execution time over 100-sample source-to-target queries are reported.**

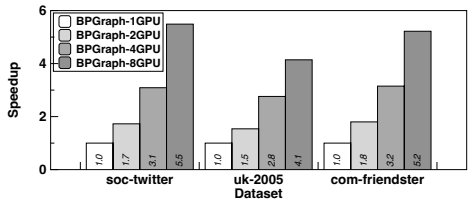



**Figure 11: Scale-up scalability evaluation of multi-GPUs on the three large datasets (soc-twitter, uk-2005, com-friendster).**

overhead than the random sampling operation, and the computation is always pushed on the same path collections. Furthermore, the results of distance-constraint reachability and expected shortest distance metrics are filtered from reachability-based results using additive constraints based on all identified possible paths. Thus, the calculation of different distance metrics over paths are trivial compared to the sampling workload.

## 5.6 Scale-Up Scalability over Multiple GPUs

To observe the effect of scaling the uncertain graph processing procedure of our system BPGraph from one GPU to multiple GPUs, we evaluate BPGraph with the large graphs, i.e., soc-twitter, uk-2005 and com-friendster, with up to 8 GPUs and illustrate the speedups in Figure 11. The performance does scale well as we add more GPUs. Specifically, the performance of multi-GPU execution on *soc-twitter* achieves 1.7× speedup from 1 GPU to 2 GPUs, and 5.5× speedup from 1 GPU to 8 GPUs. Although the evaluated performance does not scale linearly with more GPUs due to the limitation of PCI-E bandwidth, BPGraph still achieves a good scalability due to process larger dataset uk-2005 and com-friendster using more GPUs. We observe that during BPGraph processes larger uncertain graphs with more amount of edges, adding more GPUs produces much greater speedup and larger reductions in processing time.

This speedup of adding more GPUs greatly depends on the dependency of partitioned storage of consecutive edge array. BPGraph reduces the communication overhead by using vertex status array to mark the primary/secondary vertices for value synchronization, which minimizes unnecessary GPU communication traffic. On the uk-2005 dataset, BPGraph achieves a better scalability on multiple GPU, and takes 1498.5s on executing the 100 s-t query tasks using 8 GPUs. BPGraph achieves almost 4.2× speedup using 8 GPUs over using 1 GPU (6207s). For the larger dataset com-friendster, the result presents a best speedup of BPGraph, in which BPGraph achieves 5.2×, 3.2×, 1.8× speedup using 8 GPUs, 4GPUs and 2GPUs over one single GPU. This is because the comparatively peer-to-peer communication and cooperative device synchronization gives us the benefit of good scalability on large uncertain graph datasets on multi-GPU servers.

# 6 Related Work

We review the existing reliability query work for uncertain graphs and also discuss several GPU-accelerated system designs. Our proposed system BPGraph advances the state-of-the-art in the parallel design and implementation of uncertain graph processing.

**Uncertain Graph Processing.** Recently, several efficient processing approaches have been proposed to use either entirety sampling [13, 26, 48] or partition sampling methodology [8, 17, 37, 62]. The entirety sampling techniques have been widely studied for queries, e.g., reachability [47], k-nearest neighbors [49, 63, 67], (k, $\eta$)-core decomposition [36]. Although many optimizations, e.g., Monte Carlo sampling method [26, 48], recursive sampling method [35], and the representative selection method [47], have been developed. There still lacks a general framework to efficiently support processing large-scale uncertain graph data. On the other hand, the partition sampling methods [8, 37] utilize compact and partitioned data structures, which generate a small uncertain subgraph for querying purposes and answers the reliable results with fewer samples. However, these state-of-the-art methods still face sub-optimal performance due to sampling every edge, which significantly toggle down the entire performance.

**GPU-based Graph Processing.** The works on high performance and scalable GPU-accelerated graph algorithm optimization [6, 9, 20, 22, 40, 60] and system design [14, 18, 45, 57, 58] have been a hot topic by exploiting powerful computation ability of accelerators. Among these GPU-based systems, Medusa [65] proposes to simplify the programming API for GPU-based graph algorithms. CuSha [28] proposes G-Shard to improve the inefficiency of warp execution on CSR-formatted graphs and concatenated windows to address the non-coalesced memory access problem. WS-VR [27] provides warp segmentation method to enhance the GPU device utilization on dealing with irregular structural graphs, and also scale the system to multiple GPUs via a vertex refinement to reduce unavailable data transfer between GPUs via the PCIe bus. Gunrock [59] proposes a new vertex-centric programming abstraction built upon the parallel operations on a vertex or edge frontier, as well as it supports to scale to multiple GPUs via optimizing the traversal direction and GPU memory allocation. Groute [4] proposes an asynchronous processing model for scheduling computation and communication over multiple devices on a single node. Groute captures all the irregular parallelism via pushing computation on each individual vertex and improve the communication around GPUs. GraphReduce [54] is the first system to support out-of-core graph processing on GPU of a single node, which proposes streaming shard partition and hybrid vertex-centric and edge-centric parallelism model to achieve iterative large graph processing in GPUs.

Distinct from the above system design, BPGraph based on the design paradigm of uncertain graph processing, not only significantly reduces the main bottleneck from massive sampling operations of the state-of-the-art uncertain graph processing framework via providing a novel path sampling, but also considers the high performance of scaling GPU accelerator by exploiting several novel strategies to handle SIMT-aware parallel path generation, traversal, sampling combination and synchronization.

## 7 Conclusion and Future Work

In this work, we propose BPGraph, a novel multi-accelerator based framework for efficiently processing uncertain graph analytics to tackle the challenges we have observed from the state-of-the-art techniques: low computation efficiency, large memory overhead, lack of support for modern accelerators with massive parallelism, and hard for users to simply write highly-efficient uncertain graph analytics. At its core, BPGraph is integrated with a newly proposed runtime path sampling technique to identify unnecessary edges for sampling given a certain problem, resulting in drastic reduction in the overall computation. BPGraph provides general support for users to write a wide range of uncertain graph applications without dealing with the low-level complexity. Results on real-world uncertain graph applications show that BPGraph can achieve up to *43*× (*26*× on average) speedup over the state-of-the-art frameworks, and scales well with increasing number of GPUs.

## Acknowledgment

We would like to thank our shepherd and all anonymous reviewers. This work is supported by University of Sydney (USYD) faculty startup funding, SOAR faculty fellowship and Australian Research Council (ARC) DP210101984. It is also supported by the National Science Foundation of China (NSFC) Grant No. 62002350, Key-Area Research and Development Program of Guangdong Province Grant No. 2019B010154004, and Tencent Youtu Lab. Hang Liu was in part supported by the NSF CRII Award No. 2000722 and CAREER Award No. 2046102.

## References

[1] http://law.di.unimi.it/webdata/. LAW web dataset. (http://law.di.unimi.it/webdata/).

[2] Eytan Adar and Christopher Re. 2007. Managing uncertainty in social networks. *IEEE Data Eng. Bull.* 30, 2 (2007), 15–22.

[3] Michael O Ball. 1986. Computational complexity of network reliability analysis: An overview. *IEEE Transactions on Reliability* 35, 3 (1986), 230–239.

[4] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 235–248.

[5] Paolo Boldi, Francesco Bonchi, Aris Gionis, and Tamir Tassa. 2012. Injecting uncertainty in graphs for identity obfuscation. *arXiv preprint arXiv:1208.4145* (2012).

[6] Federico Busato and Nicola Bombieri. 2015. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2015), 1826–1838.

[7] Yurong Cheng, Ye Yuan, Lei Chen, and Guoren Wang. 2015. The reachability query over distributed uncertain graphs. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 786–787.

[8] Yurong Cheng, Ye Yuan, Lei Chen, Guoren Wang, Christophe Giraud-Carrier, and Yongjiao Sun. 2016. Distr: A distributed method for the reachability query over large uncertain graphs. *IEEE Transactions on Parallel and Distributed Systems* 27, 11 (2016), 3172–3185.

[9] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. 2014. Efficient multi-GPU computation of all-pairs shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 360–369.

[10] Talya Eden, Shweta Jain, Ali Pinar, Dana Ron, and C. Seshadhri. 2018. Provable and Practical Approximations for the Degree Distribution Using Sublinear Graph Samples. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW'18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 449–458. https://doi.org/10.1145/3178876.3186111

[11] Talya Eden, Amit Levi, Dana Ron, and C. Seshadhri. 2015. Approximately Counting Triangles in Sublinear Time. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. 614–633. https://doi.org/10.1109/FOCS.2015.44

[12] Talya Eden, Dana Ron, and C. Seshadhri. 2020. Faster Sublinear Approximation of the Number of k-Cliques in Low-Arboricity Graphs. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms* (Salt Lake City, Utah) *(SODA'20)*. Society for Industrial and Applied Mathematics, USA, 1467–1478.

[13] George S Fishman. 1986. A comparison of four Monte Carlo methods for estimating the probability of st connectedness. *IEEE Transactions on reliability* 35, 2 (1986), 145–155.

[14] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 1–6.

[15] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 121–131.

[16] Anil Gaihre, Da Zheng, Scott Weitze, Lingda Li, Shuaiwen Leon Song, Caiwen Ding, Xiaoye S Li, and Hang Liu. 2021. Dr. Top-k: delegate-centric Top-k on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[17] Xiulian Gao and Yuan Gao. 2013. Connectedness index of uncertain graph. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 21, 01 (2013), 127–137.

[18] Tong Geng, Tianqi Wang, Chunshu Wu, Chen Yang, Shuaiwen Leon Song, Ang Li, and Martin Herbordt. 2019. LP-BNN: Ultra-low-latency BNN inference with layer parallelism. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 9–16.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.

[20] Pawan Harish and PJ Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High-Performance Computing*. Springer, 197–208.

[21] Mark Harris and Kyrylo Perelygin. 2017. Cooperative groups: Flexible CUDA thread programming. *NVIDIA Developer Blog* (2017).

[22] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 267–276.

[23] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 78–88.

[24] 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.

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

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

[27] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 39–50.

[28] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 239–252.

[29] Hyeongsik Kim, Abhisha Bhattacharyya, and Kemafor Anyanwu. 2019. Semantic Query Transformations for Increased Parallelization in Distributed Knowledge Graph Query Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/3295500.3356212

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web* (Raleigh, North Carolina, USA). ACM, New York, NY, USA, 591–600.

[31] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 225–236.

[32] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-consolidation: A novel execution model for gpus. In *Proceedings of the 2018 International Conference on Supercomputing*. 53–64.

[33] Ang Li, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda, and Henk Corporaal. 2016. X: A comprehensive analytic model for parallel machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 242–252.

[34] Ang Li, Shuaiwen Leon Song, Akash Kumar, Eddy Z Zhang, Daniel Chavarría-Miranda, and Henk Corporaal. 2016. Critical points based register-concurrency

autotuning for GPUs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1273–1278.

[35] Rong-Hua Li, Jeffrey Xu Yu, Rui Mao, and Tan Jin. 2015. Recursive stratified sampling: A new framework for query evaluation on uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering* 28, 2 (2015), 468–482.

[36] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92.

[37] Silviu Maniu, Reynold Cheng, and Pierre Senellart. 2017. An Indexing Framework for Queries on Probabilistic Graphs. *ACM Transactions on Database Systems* 42, 2 (2017).

[38] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R Alam, Thomas C Schulthess, and Torsten Hoefler. 2016. A PCIe congestion-aware performance model for densely populated accelerator servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 63.

[39] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25.

[40] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.

[41] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-performance and scalable GPU graph traversal. *ACM Transactions on Parallel Computing* 1, 2 (2015), 14.

[42] Marco Minutoli, Prathyush Sambaturu, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Anil Vullikanti. 2020. Preempt: Scalable Epidemic Interventions Using Submodular Optimization on Multi-GPU Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 55, 15 pages.

[43] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[44] Evdokia Nikolova, Matthew Brand, and David R Karger. 2006. Optimal Route Planning under Uncertainty.. In *Icaps*, Vol. 6. 131–141.

[45] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2015. Multi-GPU graph analytics. *arXiv preprint arXiv:1504.04804* (2015).

[46] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[47] 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 the 2014 ACM SIGMOD international conference on management of data*. 967–978.

[48] 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.

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

[50] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

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

[52] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[53] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In *High Performance Computing*. Springer International Publishing, Cham, 97–119.

[54] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 28.

[55] Philip Taffet and John Mellor-Crummey. 2019. Understanding Congestion in High Performance Interconnection Networks Using Sampling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 43, 24 pages. https://doi.org/10.1145/3295500.3356168

[56] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. 2016. Combating the reliability challenge of GPU register file at low supply voltage. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 3–15.

[57] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[58] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

[59] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 11.

[60] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D Owens. 2015. Performance characterization of high-level programming models for GPU graph analytics. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 66–75.

[61] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.

[62] 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 *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 64–75.

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

[64] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-Based Iterative Directed Graph Processing System on Multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 601–614.

[65] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.

[66] Jian Zhou, Fan Yang, and Ke Wang. 2014. An inverse shortest path problem on an uncertain graph. *Journal of Networks* 9, 9 (2014), 2353.

[67] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2017. Towards efficient top-k reliability search on uncertain graphs. *Knowledge and Information Systems* 50, 3 (2017), 723–750.

[68] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX Annual Technical Conference*. 375–386.

[69] Zhaonian Zou, Faming Li, Jianzhong Li, and Yingshu Li. 2017. Scalable Processing of Massive Uncertain Graph Data: A Simultaneous Processing Approach. In *IEEE International Conference on Data Engineering*.

[70] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 649–652.

# A Accuracy Analysis of Path-Sampling Methodology

This section provides theoretically analysis on the accuracy of uncertain graph methods.

Consider an uncertain graph $\mathbb{G} = (V, E, P)$ with $|V| = n$ and $|E| = m$, where $V$ and $E$ denote the set of nodes and edges respectively. $P$ is a set of probabilities representing the likelihoods of the existence of edges, i.e., $p_e$ denotes the probability of $e \in E$. The existence of an edge is independent with each other. Let $G = (V_G, E_G)$ be a possible graph which is obtained by sampling each edge e in $\mathbb{G}$ following the probability $p_e$. Obviously, $V_G = V$, $E_G \subset E$, and the probability of $G$ is given by

$$\Pr[G] = \prod_{e \in E_G} p_e \prod_{e \in E \setminus E_G} (1 - p_e)$$

Taking into account that the accuracy of results are dependent on the number of sample worlds, we further theoretically analyze the accuracy achieved by recent sampling methodologies, entirety, partition and our path sampling methodologies. There are two parts that cause accuracy loss: *(1) the choice of sampling method*; and *(2) the choice of decomposition and preprocessing methods*.

Referring to *(1) the choice of sampling method*, since sampling is a technique for approximate query processing, it will lose information while accelerating the querying process, and less sampling time leads to larger errors in the reliable results. To achieve a theoretical estimation accuracy, the *Chernoff* bound is widely applied to determine the number of possible worlds in uncertain graph literature. Given an uncertain graph $\mathbb{G}$, a distance function $d$, and a pair of nodes $s$ and $t$, the accuracy of estimating the value of $d(s,t)$ by MC sampling can be well guaranteed. To achieve an error rate of $\epsilon > 0$ with a failure probability of $\sigma > 0$, i.e., the number of samples needed is: $g(\mathbb{G}, s, t, \epsilon, \delta) = \max\left\{\frac{3}{\epsilon^2 d'(s,t)}, \frac{\phi(\mathbb{G})^2}{2\epsilon^2}\right\} \cdot \ln\left(\frac{2}{\delta}\right)$, where $d'(s,t)$ is the estimated value of $d(s,t)$, and the function $\phi(\mathbb{G}) = \max_{(s,t) \in V \times V} d(s,t)$ is the diameter of $\mathbb{G}$.

In practice, one usually focuses on finding the pairs with a given threshold $\rho$. Note that in general $\rho$ is not too small, and thus we have $\frac{\phi(\mathbb{G})^2}{2\epsilon^2} \geq \frac{3}{\epsilon^2 \rho}$. Therefore, the number of needed samples is computed as: $g(\mathbb{G}, \epsilon, \delta) = \frac{\phi(\mathbb{G})^2}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right)$.

Referring to *(2) the choice of decomposition and partition methods*, there exists inaccurate impacts in the generated uncertain subgraphs due to information loss during graph decomposition (e.g., deleting vertices or edges), leading to errors in reliability results.

*Entirety Sampling*. Entirety sampling estimates the reliable results from the full uncertain graph, reporting source-to-target reachability as 1 or 0 in each sample. Because the entire number of possible world are generated, the entirety sampling methodology without deleting any structures ensures lossless reliability results.

*Partition Sampling*. Partition sampling methodologies use decomposition methods that result in losing information, such as, indexing tree index structures with $width > 2$ (illustrated in Section 2.2). The uncertain subgraph distilling from index searching becomes inaccurate, resulting in reliable errors. When $width = 2$, the tree is a binary tree that ensures full connection between triplets without cutting any edges [37], resulting in lossless results. The partition sampling would then be accuracy lossy with bound via building full connected index tree.

*Path Sampling*. Because of the non-redundant generated structures, path sampling is efficient on achieving reliable results. First, path sampling improves sampling efficiency by removing the large overhead of generating massive possible worlds. As a result, the sampling error is reduced, alleviating the impact from sampling insufficient amount of possible worlds. Second, path sampling is faster than entirety and partition sampling for executing uncertain graph applications because it eliminates redundant sampling overhead. Table 6 depicts the comparison of recent work on sampling methods.

**Table 6: Sampling Method Comparison.**

| Sampling Method | Space | Time | Query Accuracy |
|---|---|---|---|
| Monte Carlo | Quadratic | Linear | Lossless |
| BitEdge | Quadratic | Linear | Lossless |
| ProbTree | Quadratic | Linear | Lossy (with bound) |
| DistR | Linear | Linear | Lossy (with bound) |
| BPGraph | Linear | Linear | Lossless |