uBlade: Efficient Batch Processing for Uncertain Graph Queries

SIYUAN YAO, Singapore Management University, National University of Singapore, Singapore
YUCHEN LI, Singapore Management University, Singapore
SHIXUAN SUN, Shanghai Jiao Tong University, China
JIAXIN JIANG, National University of Singapore, Singapore
BINGSHENG HE, National University of Singapore, Singapore

The study of uncertain graphs is crucial in diverse fields, including but not limited to protein interaction analysis, viral marketing, and network reliability. Processing queries on uncertain graphs presents formidable challenges due to the vast probabilistic space they encapsulate. While existing systems employ batch processing to address these challenges, their performance is often compromised by the suboptimal selection of parallel graph traversal methods, the excessive costs in random number generation, and additional sampling-loads intrinsic to batch processing. In this paper, we introduce uBlade, an efficient batch-processing framework for uncertain graph queries on multi-core CPUs. uBlade utilizes the work-efficient graph traversal, achieving superior parallelism in the batch processing model. Additionally, our Quasi-Sampling technique reduces the random number generation cost by a factor of B, with $O(B)$ denoting the batch size. We further examine the extra sampling-load resulting from batch processing and introduce an efficient strategy to reorder possible worlds, minimizing this associated overhead. Through comprehensive evaluations, we showcase that uBlade achieves up to two orders of magnitude speedups against the state-of-the-art CPU and GPU-based solutions.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Networks** → *Network reliability*.

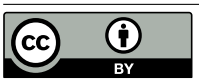
Additional Key Words and Phrases: Uncertain Graphs, Graph Algorithms, Parallel Programming

ACM Reference Format:
Siyuan Yao, Yuchen Li, Shixuan Sun, Jiaxin Jiang, and Bingsheng He. 2024. uBlade: Efficient Batch Processing for Uncertain Graph Queries. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 179 (June 2024), 24 pages. https://doi.org/10.1145/3654982

1 INTRODUCTION

Graphs play an indispensable role in representing complex relationships among entities. Yet, in numerous contexts, the interconnections between nodes are often probabilistic. For instance, in biological networks like protein-protein interactions, experimental noise means edges carry uncertain values, indicating interaction likelihoods [28]. Social networks exhibit similar uncertainty, representing the potential influence between individuals as seen in viral marketing studies [34]. In mobile ad-hoc networks, the transient connectivity translates to uncertainty in data packet transmission success [16]. To model these probabilistic connections, the notion of the *uncertain graph* emerges [3, 25, 26].

Authors' addresses: Siyuan Yao, Singapore Management University, and National University of Singapore, Singapore, siyuanyaodd@gmail.com; Yuchen Li, Singapore Management University, Singapore, yuchenli@smu.edu.sg; Shixuan Sun, Shanghai Jiao Tong University, China, sunshixuan@sjtu.edu.cn; Jiaxin Jiang, National University of Singapore, Singapore, jxjiang@nus.edu.sg; Bingsheng He, National University of Singapore, Singapore, hebs@comp.nus.edu.sg.



ACM 2836-6573/2024/6-ART179
https://doi.org/10.1145/3654982

In uncertain graphs, edges between nodes are characterized by probability values that indicate their likely existence. Aligned with prevalent literature, we embrace the *possible-world* semantics, wherein each world is defined by independently assessing the existence of edges based on their probabilities [1, 19, 25, 33, 40]. Central to uncertain graph analysis are two fundamental queries on the probability distribution of the possible worlds: a) *reliability* [7, 8, 19]: determine the probability that two given nodes are reachable; b) *shortest distance* [21, 42, 54]: identifies the distance between two given nodes with the maximum likelihood of being the shortest. Employing these two queries underpins numerous algorithms to drive downstream applications, such as kNN [6, 31, 42], influence estimation [1, 35], and network clustering [21, 27].

Challenges. Deriving the exact solutions for the fundamental queries is #P-hard and requires aggregating results across all 2^m possible worlds on an uncertain graph with m edges [48]. Consequently, existing works resort to approximated solutions that employ Monte-Carlo methods for sampling possible worlds when processing uncertain graph queries [40]. However, ensuring accuracy demands extensive processing over a large number of possible worlds. This computational demand drives the design of uncertain graph processing systems [29, 51, 56], which tap into parallelism for querying independently sampled possible worlds and leverage *batch processing* for better cache locality. Despite these recent efforts, the performance of existing systems is still unsatisfactory due to the following reasons.

First, existing systems fail to balance sampling-load and parallelism when picking the graph traversal algorithm. The state-of-the-art, Sage [29], utilizes the Bellman-Ford algorithm to traverse the possible worlds to maximize parallelism, despite its heavy computational load. In contrast, the Dijkstra algorithm offers greater efficiency in terms of sampling-load. However, this choice is constrained by limited parallelism, owing to the sequential characteristics of the Dijkstra algorithm.

Second, current systems grapple with excessive sampling overheads, a known bottleneck for uncertain graph query processing [51]. While Sage offers a promising *Deterministic Sampling* approach, circumventing the expensive materialization of multiple possible worlds through on-demand verification of edge existence using fixed random seeds, the approach remains suboptimal. This is because one needs to repeatedly generate a large amount of random numbers to determine the presence of each visited edge when traversing each possible world.

Third, the batch processing model employed in existing systems brings additional sampling-load. In the batch mode, active nodes from multiple possible world traversals are jointly processed. Moreover, once node u is visited in a traversal iteration, the existence of all its neighbors must be verified for **every** possible world in the batch where u has been or is being visited. This batching approach naturally imposes a greater sampling-load than processing each possible world independently, *a.k.a.* the non-batch mode. Surprisingly, existing literature has overlooked this extra overhead introduced by batching in uncertain graph processing.

Contribution. To address these challenges, we introduce uBlade, an efficient batch processing framework for uncertain graph queries. Our system distinguishes itself from prior studies by introducing the following novel optimizations.

Work-Efficient Batch Parallelism. In uBlade, we employ Δ-stepping to achieve work-efficient parallelism for uncertain graph traversal. Δ-stepping is a work-efficient technique originally introduced to enhance the parallelism of single graph traversals [39]. However, its potential for uncertain graph processing, particularly within the batch processing model, has remained largely unexplored in existing literature. Compared to the traversal methods commonly used in existing uncertain graph processing systems, namely Bellman-Ford [2] and Dijkstra [12], our approach consistently outperforms them, striking a balance between parallelism and work efficiency.

Quasi-Sampling. We introduce a novel Quasi-Sampling approach to address the sampling bottleneck for uncertain graph processing. In contrast to Deterministic Sampling, which needs a random number generation for every edge in *each* possible world, our approach only requires **three** random numbers to determine an edge's presence across all B possible worlds in a batch, reducing the sampling cost by a factor of $O(B)$. This efficiency is achieved by uniformly selecting possible worlds in the batch where the edge is present. In addition to better efficiency achieved due to lower sampling cost, our experiments show that Quasi-Sampling converges faster with lower variance.

Possible World Reordering. We analyze the likelihood of incurring a substantial additional sampling-load for batch processing of uncertain graph queries. The analysis reveals that the likelihood could explode for a larger batch size due to more distinct paths in different possible worlds. Motivated by the analysis, we propose a novel reordering optimization. The idea is to execute a trial run of all possible worlds to get the 2-hop neighbors, serving as a proxy for traversal patterns. We then reorder similar possible worlds into processing batches. By devising a novel *post-reorder* Quasi-Sampling, we recover their traversal status before reordering and employ *phase transition* to reduce additional sampling-load within each batch.

Results. Through experiments spanning six different applications on various datasets, uBlade delivers up to two orders of magnitude speedups compared with the state-of-the-art baselines.

Paper Organization. Section 2 provides the background of uncertain graph processing. Section 3 presents the computational framework of uBlade and we explore the optimal traversal algorithm for work-efficient parallelism in Section 4. Quasi-Sampling is proposed in Section 5, while the reordering optimization is detailed in Section 6. Experimental evaluations are discussed in Section 7, with related works and conclusions in Sections 8 and 9, respectively.

2 PRELIMINARIES

2.1 Uncertain Graph Queries

Uncertain Graph. An *uncertain graph* is defined as $\mathcal{G} = (V, E)$, where V and E represent the node and edge sets, respectively. For each edge $e \in E$, $w(e)$ denotes its non-negative weight/distance, and $p(e)$ is its existence probability. Following the literature, we assume edge probabilities to be independent [1, 19, 25, 34, 40].

Possible World. An uncertain graph $\mathcal{G}$ encapsulates $2^{|E|}$ possible worlds. A *possible world* $G_i = (V, E_i)$ represents a specific instance of the uncertain graph $\mathcal{G}(V, E, p)$, where $E_i \subseteq E$ and $i \in [1, 2^{|E|}]$. The probability of observing a possible world G_i is denoted as:

$$\mathtt{Pr}(G_i) = \prod_{e \in E_i} p(e) \prod_{e \in E \setminus E_i} (1 - p(e))$$

Based on the possible world semantics, we introduce two fundamental queries: *reliability* and *shortest distance* that support vast applications for uncertain graph analytics [6, 25, 26, 31, 42].

Reliability. We introduce an *indicator function* $\mathbb{I}_i(s, t)$ for $s, t \in V$ on a possible world G_i of $\mathcal{G}$. Specifically, $\mathbb{I}_i(s, t) = 1$ if there exists a path from s to t in G_i, and 0 otherwise. The reliability from s to t in $\mathcal{G}$ is defined as $R(s, t) = \sum_{i=1}^{2^{|E|}} \mathbb{I}_i(s, t)\mathtt{Pr}(G_i)$.

Shortest Distance. Unlike deterministic graphs, the shortest distance in uncertain graphs can vary across possible worlds [21, 42]. Define an indicator function $\mathbb{I}_i(s, t, d)$ for $s, t \in V$, where d denotes a specific distance value. The function outputs $\mathbb{I}_i(s, t, d) = 1$ if the weight of the shortest path from s to t equals d in G_i, and 0 otherwise. The probability that d represents the shortest distance from s to t, computed as $D(s, t, d) = \sum_{i=1}^{2^{|E|}} \mathbb{I}_i(s, t, d)\mathtt{Pr}(G_i)$. The shortest distance from s to t on $\mathcal{G}$ is given by $D(s, t) = \arg\max_d D(s, t, d)$.

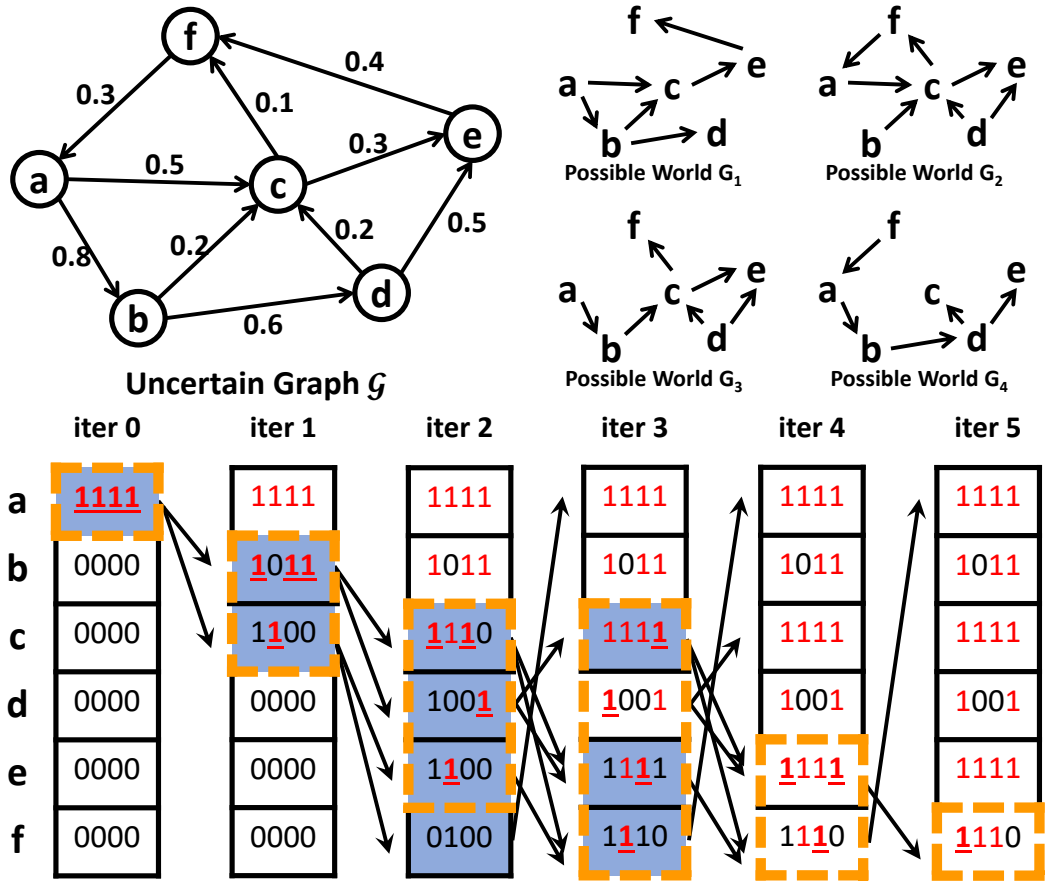


Fig. 1. Comparison of the traversal pattern of Dijkstra and Bellman-Ford on four possible worlds.

Monte-Carlo Sampling. Computing exact solutions for both reliability and shortest distance queries is known to be #P-hard [21, 48]. As such, existing works [8, 38, 42, 44, 55] resort to Monte-Carlo sampling, i.e., to evaluate the query on a subset of random possible worlds. Consider the reliability query. When given an uncertain graph $\mathcal{G}$, one can generate θ possible worlds as $\{G_1, G_2, \ldots, G_\theta\}$. The reliability can then be approximated as $\hat{R}(s,t) = \sum_{i=1}^{\theta} \frac{\mathbb{I}_i(s,t)}{\theta}$. $D(s,t)$ can also be approximated with $\hat{D}(s,t)$ by generating possible world samples.

Example 2.1. In Figure 1, we generate four possible worlds (G_1-G_4) from an uncertain $\mathcal{G}$ with 6 nodes and 10 edges. b is reachable by a in possible worlds G_1, G_3 and G_4. Thus, the reliability $\hat{R}(a,b) = \frac{3}{4} = 75\%$. For the shortest distance from a to f, we have $\hat{D}(a,f,2) = \frac{1}{4} = 25\%$ (on world G_2) and $\hat{D}(a,f,3) = \frac{2}{4} = 50\%$ (on worlds G_1 and G_3), resulting $D(a,f) = 3$ since $\hat{D}(a,f,3) > \hat{D}(a,f,2)$.

2.2 Batching of Possible World Traversals

Batch Processing. Instead of independently traversing each possible world, a batch of possible worlds are grouped and their active nodes are processed jointly in each traversal iteration. This joint processing enhances cache efficiency for random edge accesses.

Table 1. Comparisons of traversal algorithms with varying batch sizes on biomine. The best performances are boldfaced.

	Dijkstra				**Bellman-Ford**				**Δ-stepping**			
Batch size	1	20	50	100	1	20	50	100	1	20	50	100
Running time (s)	**22.71**	57.11	52.53	57.87	47.77	**37.05**	37.31	38.62	14.25	**11.26**	13.12	13.20
Workloads (x10^6)	**23.60**	505.72	570.21	625.71	**214.46**	362.97	371.64	417.42	**99.03**	175.35	178.30	187.92
LLC loads (x10^9)	5.91	2.56	1.75	**1.21**	6.05	2.43	1.67	**1.18**	4.56	1.84	1.47	**1.21**
LLC misses (x10^9)	0.43	0.33	0.33	**0.32**	0.21	0.17	0.15	**0.12**	0.20	0.16	0.15	**0.14**

Example 2.2. The iteration diagram in Figure 1 illustrates batch traversal using either Dijkstra or Bellman-Ford. Take batched Bellman-Ford as an example. The traversal starts from node a on all four possible worlds (shown by bit vector 1111 in Iteration 0). Blue boxes indicate nodes visited in each iteration. In Iteration 0, it jointly expands to neighbors of a based on their existence in each possible world. Since (a, b) exists on G_1, G_2, and G_3, and (a, c) exists on G_1, G_2, the active nodes in Iteration 1 are b and c indicated by the bit vectors 1011 and 1100, respectively. The process unfolds until no active nodes in any world, allowing batched Bellman-Ford to terminate at Iteration 4.

Sampling-load. To understand the performance characteristics of uncertain graph processing systems, we measure the *sampling-load* as the number of random numbers generated to determine the existence of edges on all possible worlds during the traversal process.

Example 2.3. In Iteration 1 of Figure 1, the sampling-load of batched Bellman-Ford is tied to active nodes b and c. For b, we determine the presence of (b, c) and (b, d) in three worlds, while for c, the existence of (c, e) and (c, f) is checked in two worlds. Hence, the total sampling-load for Iteration 1 is $2 \cdot 3 + 2 \cdot 2 = 10$, assuming one random number per check. When comparing the overall sampling-load of Dijkstra and Bellman-Ford in batch processing, Dijkstra proves costlier in this scenario. Its active nodes, marked by the orange-outlined boxes, continue until one iteration beyond Iteration 5.

Additional Sampling-load. The additional sampling-load of the batch processing mode (for either Dijkstra or BellmanFord) is measured against the independent processing with Dijkstra. Nodes visited during independent processing are **underlined** in Figure 1. The orange boxes refer to batch Dijkstra processing and the blue boxes refer to batch BellmanFord processing. Once a node u is activated in any possible world in the prior iteration, both batch processing methods need to sample u's outgoing edges in **all** possible worlds in the same batch. Hence, the 1s WITHOUT underlines (and their outgoing edges) in either orange or blue boxes refer to the additional sampling-load. From our example, batch traversal could experience increased sampling-loads compared to their independent counterpart. For instance, in independent processing with Dijkstra, a node and its neighbors are visited in just one world per iteration, revealing that both batched algorithms incur extra sampling-loads. In this paper, we introduce analysis and novel optimizations to address this extra cost, often neglected by existing uncertain graph processing systems.

3 uBlade FRAMEWORK

We develop uBlade as an in-memory parallel system for uncertain graph queries. Figure 2 presents the system architecture. We maintain a single copy of the uncertain graph $\mathcal{G}$ for fast in-memory accesses and the possible worlds are generated on demand without materialization. We adopt the batch processing model and each batch contains B possible worlds. Upon dispatching the batches, the batch manager groups similar possible worlds and employs phase transition to mitigate the additional sampling-loads from batching, which will be elaborated in Section 6.

uBlade can be built on top of any generic graph processing system to process a batch. For this study, we have selected Ligra [11, 45], a widely recognized graph processing system. Ligra's

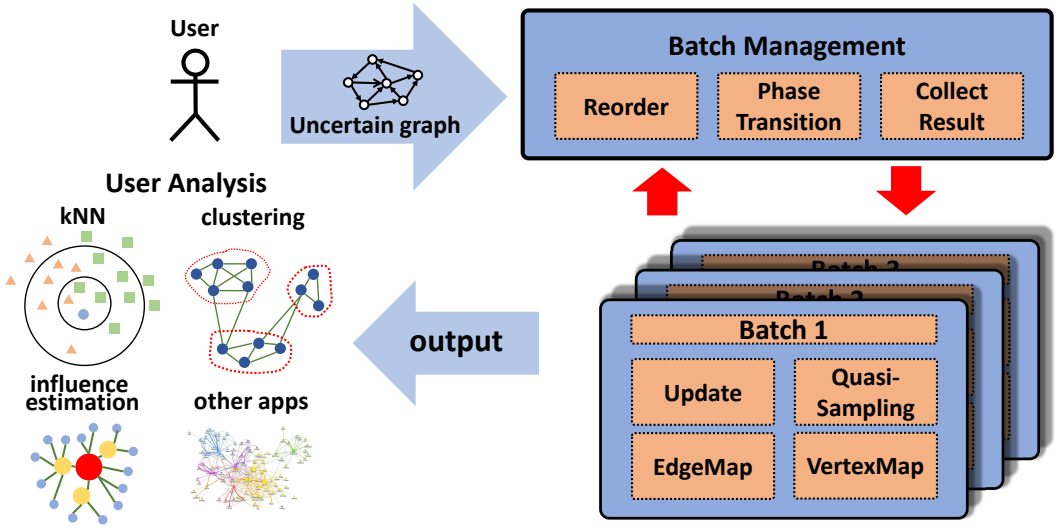


Fig. 2. The uBlade architecture.

VertexMap and EdgeMap modules offer generic support for specifying node and edge functions tailored to various graph processing tasks.

To ease the development of user analysis for uncertain graphs, uBlade provides two user-defined APIs: Update and Collect_Result. The Update API allows users to define the traversal expansion logic for an edge $e = (u, v)$ with weight w, considering its presence in a possible world indexed by *pid* within a batch. We exclude the discussion of the node update API in this paper because most uncertain graph analyses focus on edges. However, uBlade readily accommodates the node update function. Meanwhile, the Collect_Result API assists users in aggregating results for any node t based on the traversal outcomes of the batch.

Next, we present code snippets for the implementation of user-defined APIs for s-t reliability and s-t shortest distance queries with a source node s and a arget node t

s-t reliability. The implementation is depicted in Figure 3. Here, the global variable `Result` maintains the reachability information for all nodes from the source node s within each batch. To obtain reliability information for the target node t, users can collect data stored in `Result`[t] by defining the `Collect_Result()` function.

```
bool Update<bool> (u, v, w, pid){
  return true;
}
float Collect_Result<float> (t){
  int reach = 0;
  for (int i = 0;i < total_sample;i++)
    if (Result[t*B+i] != INF) reach++;
  return reach/total_sample;
}
```

Figure 3. s-t reliability.

s-t shortest distance. The implementation is illustrated in Figure 4. In this case, `Result` maintains the distance information from the source node s to each node. To gather results, users count the

number of possible worlds in which a distance from s to t is the shortest, thus obtaining the most likely shortest distance. This is accomplished by implementing the Collect_Result function.

```
bool Update (u, v, w, pid){
  if (Result[v*B+pid] == INF) return w;
  return Min(Result[v*B+pid],Result[u*B+pid]+w);
}
int Collect_Result<int> (t){
  int d = 0, reach = 0;
  vector<int> cnt(MAX_DISTANCE);
  for (int i = 0;i < total_sample;i++)
    if (Result[t*B+i] != INF){
      cnt[Result[t*B+i]]++;
      if (cnt[Result[t*B+i]] > reach){
        reach = cnt[Result[t*B+i]];
        d = Result[t*B+i];} }
  return d;
}
```

Figure 4. s-t shortest distance.

4 WORK-EFFICIENT BATCH PARALLELISM

4.1 Traversal Algorithms Profiling

At the core of existing uncertain graph systems are their traversal algorithms, with Dijkstra [12] and Bellman-Ford [2] being the predominant choices. We conduct an in-depth analysis of their respective strengths and weaknesses and elucidate their potential impact on system performance.

Settings. We profile various performance factors by varying the batch size B, including the workload and the last-level cache (LLC) efficiencies. All traversal algorithms are implemented with Ligra's VertexMap and EdgeMap functions for a fair comparison. When $B = 1$, we adopt a parallel approach by processing a possible world with one thread. Otherwise, we allocate all available threads to process each batch by jointly parallelizing VertexMap and EdgeMap of all possible worlds during each traversal iteration of the respective algorithm. The results of processing 100 possible worlds in total on the `biomine` dataset are presented in Table 1. We chose `biomine` because it is the largest dataset we have with probability values coming from real-world applications. The statistics of the dataset are shown in Table 2. `biomine` has a similar average degree to most datasets. Also, from Table 3, we can see that all systems have similar performance on `biomine` compared to other datasets. Thus we used it as a representative dataset to facilitate our discussion.

Observation 1: Improved cache efficiency with larger batch sizes. As the batch size grows, the cache efficiency gets better. This is due to the batch processing of VertexMap and EdgeMap functions during traversal iterations across all worlds. The combined processing of active nodes in all worlds cuts down random edge accesses to the uncertain graph, promoting better data locality. This aligns with findings from earlier research on concurrent graph query processing [18, 36, 47].

Observation 2: The paradox of cache efficiency and workload. While improved cache efficiency is observed, it does not invariably translate to better overall performance in uncertain graph processing, owing to the additional sampling-loads discussed in Section 2.2. Notably, both Dijkstra and Bellman-Ford algorithms exhibit sharp increases in workload with a larger batch. This surge is attributed to a side effect of batch processing: upon visiting an active node u in a possible world, random numbers must be generated to check the existence of *all* edges from u in *all* possible worlds where u *is/was* visited during the batch traversal process.

Observation 3: Inadequacy of Dijkstra and Bellman-Ford for batch processing. Neither Dijkstra nor Bellman-Ford emerges as a suitable algorithm for the *batch processing* model. While Bellman-Ford possesses higher complexity than Dijkstra in a sequential setting, Dijkstra experiences higher workloads for larger batch sizes, as evidenced in Table 1. This can be attributed to its traversal patterns, which fluctuate more significantly across different possible worlds. Consequently, when juxtaposing Dijkstra and Bellman-Ford under varied batch size settings, the performance peaks with the independent processing approach using Dijkstra ($B = 1$).

4.2 uBlade Batch Traversal

Inspired by the profiling results, we employ Δ-stepping as the traversal algorithm for uBlade.

Δ-stepping. Δ-stepping, originally formulated to bolster the parallelism of single graph traversals while safeguarding work efficiency [39], divides the nodes into buckets while traversing the graph. Each bucket encompasses nodes with tentative distances, confined within a specific range, quantized by the parameter Δ. Unlike Dijkstra's algorithm, which iteratively expands a single node, or Bellman-Ford, which processes all active nodes in an iteration, Δ-stepping iteratively processes all nodes within a bucket, thereby striking a harmonious balance between work efficiency and parallelism. Essentially, a smaller Δ aligns the approach with Dijkstra's, processing a reduced number of nodes per iteration to uphold work efficiency, while a larger Δ leans towards Bellman-Ford, processing an increased number of nodes per iteration to enhance parallelism. To the best of our knowledge, there has been no preceding investigation into the application of Δ-stepping for uncertain graph processing, especially for batch processing.

As a result, we have devised an efficient batch processing approach for possible worlds utilizing Δ-stepping and executed a comparative analysis against rival traversal algorithms, as depicted in Table 1. It is noteworthy that while Δ-stepping imposes a more substantial workload for $B = 1$ compared to Dijkstra's algorithm, it surpasses all other algorithms for $B = 20$ and beyond. This improved performance is attributed to Δ-stepping's significant sampling-load reduction during batch processing. Furthermore, it demonstrates superior cache efficiency compared to algorithms executed in an embarrassingly parallel fashion. We present the batch traversal approach of uBlade with Δ-stepping as follows:

uBlade **batch traversal (Algorithm 1).** uBlade initializes the buckets for Δ-stepping with the source node. Within each iteration, we process the next available bucket, increasing the distance value from the source node by Δ if the current bucket is empty (Line 2-5). During an iteration, EdgeMap scans all neighbors of each node in the current bucket *Bkt*. In Line 10, we employ QuasiSampling to obtain the existence information of the edge (u, v) across all possible worlds in the batch just *once*. The details of QuasiSampling will be discussed in the next Section. Following that, we update the traversal results for (u, v) for each possible world (Line 12-17), utilizing atomic compare-and-swap (CAS) operations to ensure the consistency of parallel programs. If the update is successful, the neighbor v is added to *Active*.

In the VertexMap, we process each active node v and determine its distance from the source across all possible worlds. The shortest of these distances is then used to place v in the corresponding bucket. We assume the *Result* array stores the distance information for all nodes to simplify the presentation and the distance information can be maintained separately if needed.

Note. The choice of Δ is typically chosen empirically in the literature [9–11], and we evaluate the performance of uncertain graph processing with different Δ values in Section 7.4.

Algorithm 1: uBlade Batch Traversal

```
Input: Uncertain Graph 𝒢, Source Node s, Batch ID bid
Initialize Bkts with source node s;
while Bkts has next do
    Bkt = Bkts.Next();
    Active = EdgeMap(Bkt,seed);
    VertexMap(Active, Bkts);
Function EdgeMap(Bkt, bid):
    Active = { };
    forall u ∈ Bkt in parallel do
        forall neighbor v of u in parallel do
            BatchExist = QuasiSampling(u, v, bid);
            success = false;
            forall pid ∈ {0, ..., B − 1} do
                if BatchExist[pid] then
                    idx = v · d + pid;
                    oval = Result[idx];
                    nval = Update(u, v, w(u, v), pid);
                    success = CAS(&Result[idx], oval, nval);
            add v to Active if success;
    return Active;
Function VertexMap(Active, Bkts):
    forall v ∈ Active in parallel do
        bktId = ∞;
        forall pid ∈ {0, ..., B − 1} do
            bktId = Min(bktId, Bkts.GetBkt(Result[v·B+pid]));
        Bkts.SetBkt(u, bktId);
```

5 QUASI-SAMPLING

To execute the Monte-Carlo simulation on a batch of possible worlds, the straightforward implementation materializes the edge existence information for each world independently [19, 22, 38]. However, this approach is rather memory-prohibitive when dealing with larger batch sizes on massive graphs. Consequently, Sage [29] introduced a *Deterministic Sampling* approach that circumvents the memory overhead by computing an edge's presence in a possible world with an online sampling approach.

Deterministic Sampling. To check if an edge (u, v) exists in a possible world indexed by pid, the approach utilizes an online random number generator DetminSampling(u, v, pid) where the input parameters u, v, pid jointly form the seed of the generator. In this way, the existence of different edges on different possible worlds is randomly determined while the generator always yields the same output as a means to "save" the existence information for (u, v) in possible world pid. Yet, although the approach avoids the materialization cost, it incurs higher computational costs for

generating the random numbers online. Specifically, it incurs $O(Bm)$ operations for random number generations for processing a batch of B possible worlds on an uncertain graph with m edges.

Quasi-Sampling. We introduce a novel Quasi-Sampling approach to mitigate the expensive costs associated with online random number generation in Deterministic Sampling. This approach reduces the cost from $O(Bm)$ to $O(m)$ for batch processing. In particular, our approach only necessitates *three* random numbers to capture the existence information of an edge across all possible worlds within the same batch.

Intuitively, given an edge (u, v), we identify $p(u, v) \cdot B$ possible worlds within the batch where (u, v) exists. This ensures that the probability of (u, v)'s existence aligns with $p(u, v)$. Our method of selecting these possible worlds is as follows:

$$pid_i = (start + step \cdot i) \textbf{ mod } B, \forall i = 0, \ldots, passNum - 1 \tag{1}$$

Here, $start$, $step$, and $passNum$ are the three random numbers to be generated. $start$ identifies one of the possible worlds within the batch where (u, v) exists. We then pick interleaving possible worlds, with their indices pid_i separated by $step$. To maintain unbiased sampling, $passNum$ should be equal to $p(u, v) \cdot B$ on average, *i.e.,* $\mathbb{E}[passNum] = p(u, v) \cdot B$. Further, to guarantee that we always select $passNum$ distinct possible worlds in Equation 1, we select $step$ to be a random number that is smaller than B while being a *prime* as well as coprime with B at the same time. This choice ensures no duplicates in the selected possible worlds.

Algorithm 2: QuasiSampling

Input: An edge from u to v, Batch ID bid
Output: $BatchExist$

```
1  Initialize BatchExist;
2  passNum = Floor(p(u,v) · B);
3  residual = p(u,v) · B − passNum;
4  (r, idx, start) = DetminSampling(u, v, bid);
5  if r < residual then
6      passNum++;
7  step = Coprime[idx];
8  pid = start;
9  for i ∈ {0, ..., passNum − 1} do
10     BatchExist[pid] = 1;
11     pid = (pid + step) mod B;
12 return BatchExist;
```

Algorithm 2 provides the pseudo-code for Quasi-Sampling. First, we initialize $BatchExist$ as a bit array of size B to track the presence of edge (u, v) for all possible worlds within a batch. To address the issue that $p(u, v) \cdot B$ may not yield an integer, we compute the $residual$. If the $residual$ is less than a uniformly generated random number r, we increment $passNum$ by 1 to ensure $\mathbb{E}[passNum] = p(u, v) \cdot B$. Subsequently, we generate three random numbers using Deterministic Sampling with u, v, bid as the joint seed for the batch in Line 4. r is compared with $residual$ to set $passNum$ properly. idx picks a random $step$ from the $Coprime$ array, which is computed offline to house all prime numbers under B that are also coprime with B. The $Coprime$ array is lightweight for both computation and space overheads since the batch size B is often small for a large graph, i.e., less than 1000. Finally, we pick $passNum$ possible worlds from $start$ and record the results

in *BatchExist*. Note that the seed (u, v, bid) ensures we always get the same random outcome for (u, v) in batch bid.

Correctness and Convergence. Our Quasi-Sampling approach yields *unbiased* and *independent* edge samples. The theoretical properties are established in the following theorem:

Theorem 5.1. *Let $\mathbb{I}(e, x)$ denote the event of e existing in possible world x using Quasi-Sampling. We have:*

$$\Pr[\mathbb{I}(e, x)] = p(e), \ \textbf{\textit{and}} \tag{2}$$

$$\Pr[\mathbb{I}(e_i, x), \mathbb{I}(e_j, x)] = \Pr[\mathbb{I}(e_i, x)] \cdot \Pr[\mathbb{I}(e_j, x)], \ \textbf{\textit{if}}\ e_i \neq e_j \tag{3}$$

Equation 2 ensures unbiased sampling, while Equation 3 guarantees the independent property among edges from the same possible world. Due to space limits, we give a sketch of the proof. For Equation 2, in a batch of size B, each edge is equally likely to be chosen as existing, hence $\Pr[\mathbb{I}(e, x)] = E\left(\frac{passNum}{B}\right)$. Since $E(passNum) = p(e) \cdot B$, it follows that $\Pr[\mathbb{I}(e, x)] = p(e)$. For Equation 3, $\mathbb{I}(e_i, x)$ depends on $(r_i, idx_i, start_i)$, which are determined by (u_i, v_i, bid) according to Algorithm 2. Since (u_i, v_i, bid) are independent of j, $\Pr[\mathbb{I}(e_i, x)]$ and $\Pr[\mathbb{I}(e_j, x)]$ are independent.

We conduct experiments to validate the quality of Quasi-Sampling. Figure 5 demonstrates the error convergence of Quasi-Sampling versus Monte-Carlo for reliability queries. Since reliability is #P-hard, we use the results of running 10,000 possible worlds with Monte-Carlo as the ground truth to measure the errors of both methods with varying sample sizes. We show the results of four datasets since the rest of the datasets in our experiments are too large for Monte-Carlo to process 10,000 possible worlds in a reasonable time.

While all three methods achieve convergence with more possible worlds, we have the following observations regarding their differences. First, Deterministic Sampling converges slower than Monte-Carlo. The reason behind this lies in the generation of edges according to DetminSampling(u, v, pid), where the sampling results of different edges in the same possible world are dependent on (u, v) and cause correlations. Second, Quasi-Sampling reaches convergence at a pace **quicker** than Monte-Carlo and Deterministic Sampling. This efficiency is attributed to its ability to extract possible worlds notably uniformly from the uncertain graph's distribution. Quasi-Sampling effectively selects (close to) $p(e) \cdot B$ possible worlds for each batch while maintaining regular intervals between the sampled possible worlds. In contrast, the number of possible worlds selected by Monte-Carlo and Deterministic Sampling could fluctuate for the same edge in different batches.

6 POSSIBLE WORLD REORDERING

6.1 Batching Sampling-load Analysis

Recall that the number of edge probes to check their existence determines the *sampling-load*. If we expect a sampling-load of W for traversing one possible world, then having *identical* traversal patterns across all B possible worlds in a batch leads to no additional sampling-load, i.e., $O(BW)$. In contrast, varying traversal patterns can lead to a potential additional sampling-load of up to $O(B^2W)$. This is because, in the worst-case scenario, processing one world may entail sampling-loads from the remaining $B - 1$ worlds during each iteration. Moreover, the extra work from other worlds might only involve nodes that were previously visited, rendering them unnecessary if we were to traverse all possible worlds independently Hence, the variety in traversal patterns in the batch determines the *additional* sampling-loads. Since it is infeasible to capture complex traversal patterns for arbitrary queries, we use the s-t shortest path as a proxy. That is, if we have k distinct shortest paths among all possible worlds in a batch, we incur $O(k^2W)$ extra work.

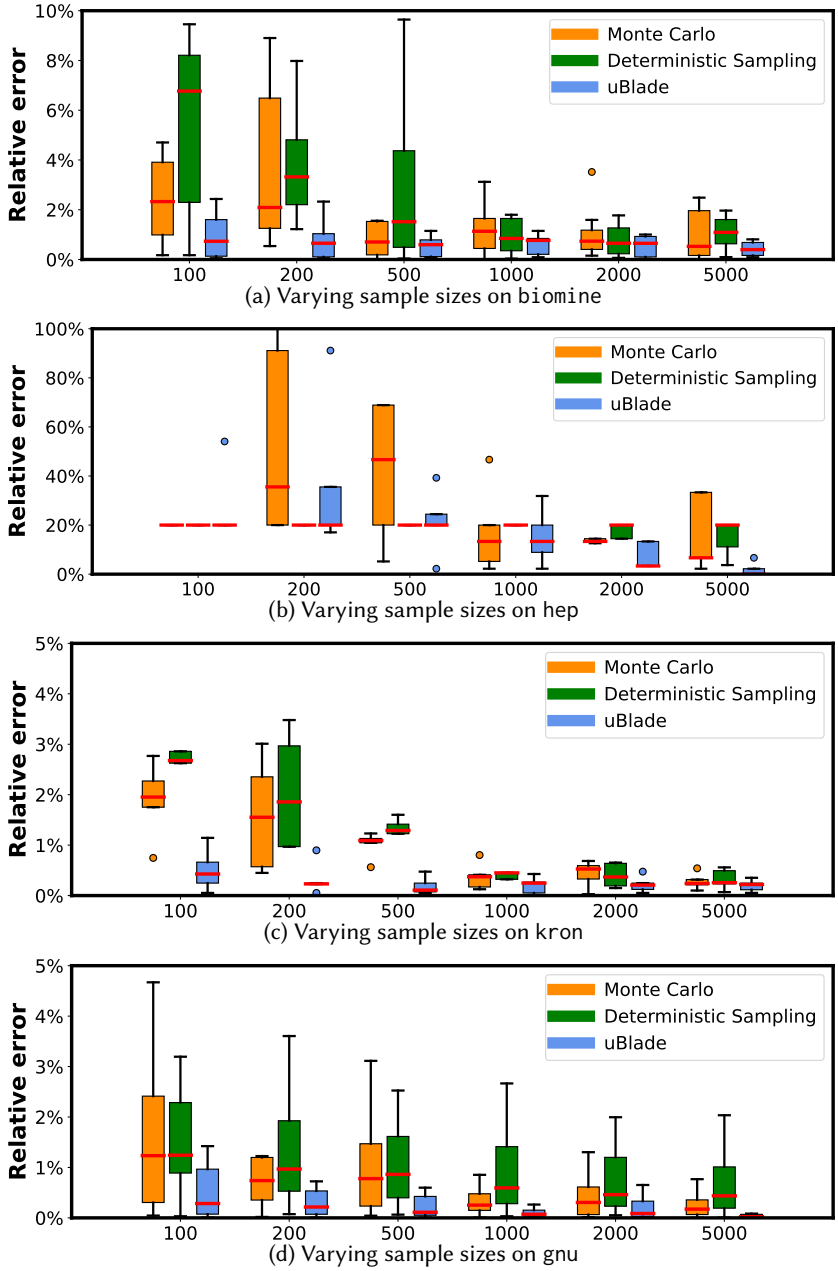


Fig. 5. Average relative errors measured by Monte-Carlo on 10,000 possible world samples.

To analyze the likelihood of k distinct paths, we start with one possible world traversal in a batch. Now by adding a new possible traversal into this batch, either a) the new traversal has the same shortest path as the first traversal, or b) the new traversal has a different shortest path. Following existing literature on random graph analysis, we assume that all edges in the uncertain graph $\mathcal{G}$ have equal existence probability, i.e., $p(e) = p$. Thus, we can bound the probability for b) as $\Pr[\text{new path}] \leq (1-p^l)$ where l denotes the longest length of all short paths on any possible world of $\mathcal{G}$. Subsequently, we derive the probability of having a new shortest path in the k^{th} possible

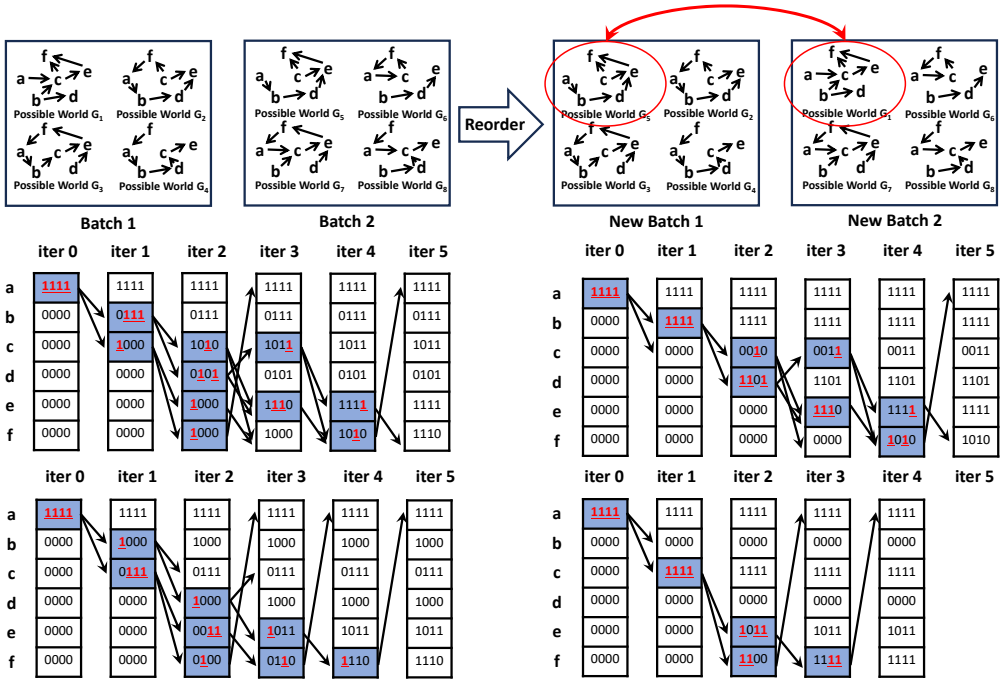


Fig. 6. Illustrations of possible world reordering. The arrow from one iter to the next implies the edge must be accessed in the original uncertain graph and needs sampling to determine if it is active in a possible world. For example, in batch 1 before reordering, node c is active at iter 3 in possible world 4. Therefore, we need to sample edges starting from node c, resulting in arrows from c to nodes e and f. As there is no edge between c and f in possible world 4, f remains unvisited at iter 4 in world 4.

world traversal, given that $k-1$ distinct short paths already exist, denoted as N_k where

$$N_k = \Pr[\text{new path} \mid k-1 \text{ distinct paths}] \leq (1-p^l)^{k-1}$$

Next, we consider the following process: starting with a batch with one possible world, followed by adding the $B-1$ possible worlds to the batch. Then, the likelihood of having k distinct shortest paths within a batch of B possible worlds is computed as

$$P_k = \binom{B-1}{k-1} \prod_{i=2}^{k} N_i \prod_{i=k+1}^{B} (1-N_k) \leq B^{k-1}(1-p^l)^{\frac{k(k-1)}{2}}(1-N_k)^{B-k}$$

The term $\binom{B-1}{k-1}$ stands for choosing $k-1$ possible worlds where each world traverses a new shortest path. Thus, the rest of the $B-k$ possible worlds do not introduce any new shortest path.

With a fixed k, the probability $P_k \to 0$ when B increases since $(1-N_k) \in (0,1)$ and $B^{k-1}(1-N_k)^{B-k} \to 0$. In other words, for a large B, k would not be small with high probability, leading to a quadratic explosion of additional sampling-loads of $O(k^2W)$.

6.2 Lightweight Reordering

Our analysis reveals differing traversal patterns across possible worlds during batch processing can introduce additional sampling-loads. To mitigate this overhead, we propose to reorder the possible worlds so that those with similar traversal patterns are grouped in batches.

Example 6.1. Figure 6 presents an example of the possible world reordering approach. There are $\theta = 8$ possible worlds to be processed with 2 batches. Based on the traversal patterns of batch processing before reordering, we swap possible world G_1 with world G_5 to form two new batches. After the reordering, the sampling-load drastically decreases for both batches with much fewer blue boxes and even a faster termination for batch 2.

Implementation Challenges. Although reordering can potentially reduce additional sampling-loads, its implementation presents challenges for two primary reasons. First, determining the traversal patterns of each possible world poses a *chicken-and-egg* problem since obtaining the patterns exactly is equivalent to obtaining the query results. Therefore, a lightweight method is required to determine the traversal patterns for each world and regroup them to form new batches. Second, the traversal pattern of a possible world *must* remain the same after assigning it to a new batch to enjoy the benefit of reordering. Otherwise, the traversal patterns are still probabilistic and incur additional sampling-loads. Hence, an efficient method must be developed to recover the edge existence information based on the random numbers generated before reordering.

2-hop Reordering. To address the first challenge, we tap into the existing literature on concurrent graph traversal [18, 36, 47], which studies the problem of multiple traversals from different source nodes on the same underlying graph. To facilitate effective batch generation, a common practice is to employ a degree-based heuristic. This heuristic arranges traversals based on the degrees of their source nodes, placing those with higher degrees in the same batches. This is because high-degree nodes often share more common neighbors on large datasets, offering improved opportunities for traversal sharing. However, the degree-ordered heuristic cannot be directly adopted for uncertain graph processing since the possible world traversals start from the same source node.

We introduce the *2-hop reordering* heuristic, which employs *two* Bellman-Ford iterations from the source node s to gather the count of 2-hop neighbors in each possible world. We then reorder the possible worlds so those with similar two-hop neighbor counts are batched together. Hence, this is a lightweight strategy to capture the traversal patterns of different possible worlds. Our experimental results demonstrate that the 2-hop reordering incurs an overhead constituting merely 4.1% of the total execution time. As detailed in Section 7.3, this offers an effective strategy for capturing traversal patterns and reducing the additional sampling-load in batch processing.

Post-reorder Quasi-Sampling. To address the second challenge of recovering the edge existence information, we develop the *post-reorder* Quasi-Sampling method. To determine the existence of an edge (u, v) in the possible world in a reordered batch, we need to obtain its old batch ID $obid$ and its possible world ID $opid$ in the old batch before the reordering process. Once $obid$ and $opid$ are available, we then use $(u, v, obid)$ as the seed to regenerate the random numbers $(r, idx, start)$ for determining if (u, v) exists.

Algorithm 3 presents the post-reorder Quasi-Sampling. $Rmap$ stores the old possible world IDs before reordering, and we obtain $obid$ and $opid$ from $Rmap$ for possible world pid in the new batch (Lines 3-4). Subsequently, we regenerate the random tuple $(r, idx, start)$ using $(u, v, obid)$ as the seed for Quasi-Sampling in Line 7. Since $step$ is coprime with B, the modular inverse of $step$ wrt. B always exists, i.e., $step^{-1}$. Further, we can efficiently recover the parameter i in Equation 1 by modulo arithmetic in Line 11. Subsequently, (u, v) exists in possible world pid if and only if $i < passNum$. Finally, we record the result in $BatchExis[pid]$.

Theorem 6.2. *For every possible world, the post-reordering Quasi-Sampling (Algorithm 3) yields edge existence information identical to that of the Quasi-Sampling (Algorithm 2) before reordering.*

Phase Transition. While post-reorder Quasi-Sampling correctly recovers edge existence information, it invokes the Deterministic Sampling for all possible worlds in the batch, as shown in Line 7

Algorithm 3: Post-reorder QuasiSampling

Input: An edge of node u and v, Reorder Map $Rmap$
Output: $BatchExist$

```
Initialize BatchExist;
for pid ∈ {0, ..., B − 1} do
    obid = Rmap[pid]/B;
    opid = Rmap[pid] mod B;
    passNum = Floor(p(u, v) · B);
    residual = p(u, v) · B − passNum;
    (r, idx, start) = DetminSampling(u, v, obid);
    if r < residual then
        passNum++;
    step = Coprime[idx];
    i = (opid − start) · step^{−1} mod B;
    if i < passNum then
        BatchExist[pid] = 1;
return BatchExist;
```

of Algorithm 3. Consequently, this negates the benefit of Quasi-Sampling detailed in Section 5, leading to an $O(Bm)$ sampling cost equivalent to that of Sage [29].

To circumvent the expensive sampling cost, we implement a phase transition approach. Initially, we execute post-reorder Quasi-Sampling for two Bellman-Ford iterations to recover the traversal process of each possible world. Subsequently, we transit to Δ-stepping traversal (Algorithm 1) and continue the traversal with the default Quasi-Sampling approach (Algorithm 2). As such, only the first two traversal iterations incur additional sampling costs. Luckily, the sampling-load associated with preliminary iterations is often negligible when compared to the entire traversal sampling-load, a common characteristic of power-law graph traversals [37, 52, 53].

7 EXPERIMENTS

7.1 Experimental Setup

Datasets. We report the datasets used in our experiment in Table 2. Datasets `netHept(hep)` and `gnutellaP2P(gnu)` were shared by [51]. Datasets `friendster(friend)`, `links-anon(anon)` and `Twitter-rv(rv)` were obtained from Stanford Network Dataset Collection [30]. Datasets `kron-logn21(kron)`, `soc-twitter(soc)`, `Web-uk-2005(uk)` and `Web-sk-2005(sk)` were obtained from Network Data Repository [43]. Dataset `biomine` was obtained from the BIOMINE project [41].

The edge probabilities in datasets `hep`, `gnu` and `biomine` come from real-world applications. The probability of edges (u, v) on `uk` and `sk` corresponds to the influence probability of u on v. Following the literature on influence analysis [4, 17, 23, 34], the probability on any (u, v) corresponds to the inverse of the out-degree of the node u. The probabilities in rest of the datasets are assigned by $p(u, v) = 1 - \exp^{(c/\mu)}$, where c is the degree of node v, using the same configuration in [24]. For `kron`, μ is 5. For `soc`, μ is 10. For the `rv`, `friend` and `anon`, μ is set to 20.

Uncertain Graph Queries. We evaluate the performance of competing systems on six queries:

- Top-k Reliability (**TopK**) [55]. Given a source s and a parameter k, find k nodes $T_k(s) = \{t_1, ..., t_k\}$ where the reliability $R(s, t_i)$ is greater or equal to $R(s, t)$ for any other node $t \in V \backslash T_k(s)$.

Table 2. Dataset statistics.

Dataset	Name	$\|V\|$	$\|E\|$	Deg_{avg}
netHept	hep	15,233	62,774	4
gnutellaP2P	gnu	62,586	147,892	2
biomine	biomine	1,508,587	32,761,889	22
kron-logn21	kron	1,544,088	91,042,012	58
soc-twitter	soc	28,504,110	531,000,244	18
Web-uk-2005	uk	39,454,748	936,364,284	24
Twitter-rv	rv	41,652,230	1,468,365,182	35
friendster	friend	65,608,366	1,806,067,135	29
Web-sk-2005	sk	50,636,151	1,949,412,601	38
links-anon	anon	52,579,682	1,963,263,821	37

- Influence Estimation (**IE**) [13]. Given a source s, compute the sum of reliability values from s to all reachable nodes.
- Network Clustering (**NC**) [27]. Given a source s, recursively dividing the network based on reliability. We follow Sage [29] and divide all networks into four clusters.
- Single Pair Shortest Distance (**SPSD**) [6]. Given a source s and a target t, find the most probable distance from s to t. We randomly select a target at least 2 hops away from the source.
- Single Source Shortest Distance (**SSSD**) [54]. Given a source s, compute the most probable shortest distances from s to all reachable nodes.
- k Nearest Neighbors (**kNN**) [42]. Given a source s and a parameter k, find k nodes $T_k(s) = t_1, ..., t_k$ where the distance $D(s, t_i)$ is less or equal to $D(s, t)$ for any other node $t \in V \backslash T_k(s)$.

For each query, we randomly select 10 source nodes and sample 100 possible worlds for each source node. For **TopK** and **kNN**, we set the parameter k as 10.

Competitors. We compare our system uBlade with the state-of-the-art graph processing systems.

- Sage [29] is the state-of-the-art CPU-based system for uncertain graph processing.
- BPGraph [51] is the state-of-the-art GPU-based system for uncertain graph processing.
- ForkGraph [37] is a cache-efficient concurrent graph processing system. It accelerates the processing of multi-source queries on deterministic graphs. We edit its edge update function with Deterministic Sampling to enable batch processing for uncertain graph queries. We partition each dataset using METIS [20] to fit each partition into the last level cache (LLC) size.
- uBlade is our proposed system with all optimizations enabled. We set $\Delta = 2$ by default. We study the impact of varying Δ on the performance for uBlade in Figure 10. The implementation of uBlade is available at https://anonymous.4open.science/r/uBlade-A8D1/README.md.

We obtained the implementations for all the systems under comparison directly from their respective authors[1][2][3]. The batch size B is set to 20 for ForkGraph and uBlade as the setting achieves the best empirical performance overall. $B = 20$ gives the best empirical performance for both ForkGraph and uBlade because they are both built on top of the Ligra framework. With a similar infrastructure, the influence of the batch size on ForkGraph and uBlade is similar.

For Sage, it always executes all possible worlds in a single batch and multiple runs of Sage instance can only produce the same set of possible worlds. For BPGraph, we encountered issues in obtaining results on our machine. Specifically, our inspections of its implementation reveal that the

[1]Sage is obtained at https://github.com/mlsys-seo/SAGE.
[2]BPGraph is obtained at https://github.com/bpgraph/bpgraph.
[3]ForkGraph is obtained at https://github.com/Xtra-Computing/ForkGraph.

Table 3. Overall performance of the compared uncertain graph systems (in seconds).

Applications	Systems	hep	gnu	biomine	kron	soc	uk	rv	friend	sk	anon
TopK	Sage	0.79	1.35	24.57	24.47	823.72	1025.74	7101.92	8618.01	1118.79	2216.92
	ForkGraph	**0.04**	0.42	16.08	18.52	536.09	246.93	2033.31	3438.09	372.19	1285.94
	uBlade	0.08	**0.12**	**2.39**	**2.54**	**71.63**	**84.01**	**472.59**	**578.71**	**99.32**	**197.39**
speedup vs. the next best		50%⇓	350%⇑	613%⇑	729%⇑	748%⇑	294%⇑	430%⇑	594%⇑	375%⇑	651%⇑
NC	Sage	3.00	4.85	98.06	104.38	2254.30	3125.13	22435.37	25910.43	3421.09	6994.07
	ForkGraph	**0.13**	1.33	21.07	24.74	643.21	313.57	2453.16	6704.35	493.76	1529.62
	uBlade	0.21	**0.33**	**3.55**	**4.55**	**95.45**	**115.73**	**514.22**	**936.90**	**138.43**	**236.19**
speedup vs. the next best		62%⇓	403%⇑	594%⇑	544%⇑	674%⇑	271%⇑	477%⇑	716%⇑	357%⇑	648%⇑
IE	Sage	0.81	1.98	27.96	30.27	913.74	1403.70	9569.17	8891.02	917.09	2080.37
	ForkGraph	**0.01**	0.32	13.20	34.95	527.45	260.07	2791.84	3609.21	382.31	1237.81
	uBlade	**0.01**	**0.11**	**2.33**	**3.13**	**65.44**	**90.26**	**511.47**	**575.78**	**70.75**	**180.02**
speedup vs. the next best		0%	291%⇑	567%⇑	967%⇑	806%⇑	288%⇑	546%⇑	627%⇑	540%⇑	688%⇑
SSSD	Sage	0.72	1.38	24.13	25.23	814.89	1020.41	7095.31	8655.12	1123.35	2241.69
	ForkGraph	**0.04**	0.41	16.22	18.20	532.17	243.71	2029.40	3427.41	370.42	1290.03
	BPGraph	1.2	2.7	-	20.8	325.1	894.2	-	-	-	-
	uBlade	0.08	**0.12**	**2.46**	**2.62**	**70.30**	**85.51**	**474.12**	**578.03**	**101.89**	**198.92**
speedup vs. the next best		50%⇓	342%⇑	659%⇑	695%⇑	462%⇑	285%⇑	428%⇑	593%⇑	364%⇑	649%⇑
SPSD	Sage	0.84	2.03	26.67	31.99	937.26	1452.61	9631.54	8903.57	899.12	2113.84
	ForkGraph	**0.01**	0.16	7.93	29.65	318.33	183.24	1945.65	2933.45	222.18	864.19
	BPGraph	2.4	4.3	-	30.6	486.3	607.6	-	-	-	-
	uBlade	**0.01**	**0.01**	**0.59**	**0.67**	**4.20**	**29.25**	**81.57**	**55.88**	**28.41**	**129.93**
speedup vs. the next best		0%	1600%⇑	1344%⇑	4425%⇑	7579%⇑	628%⇑	2385%⇑	5250%⇑	782%⇑	665%⇑
kNN	Sage	0.72	1.35	25.23	26.76	820.65	1024.26	7096.10	8647.71	1124.53	2232.03
	ForkGraph	**0.04**	0.46	17.85	19.81	540.04	247.79	2041.21	3456.86	375.53	1287.11
	BPGraph	2.8	6.1	-	138.0	1245.2	1501.2	-	-	-	-
	uBlade	0.08	**0.14**	**2.91**	**2.96**	**72.39**	**88.61**	**474.30**	**579.29**	**102.11**	**197.83**
speedup vs. the next best		50%⇓	329%⇑	613%⇑	669%⇑	746%⇑	280%⇑	430%⇑	597%⇑	368%⇑	651%⇑

number of paths precomputed in their implementations often exploded and cannot be terminated within a reasonable time. Despite reaching out to its authors, we did not receive any feedback. Thereby, we only compare the performance numbers reported in [51] on the same query types and the same settings. Hence, the results for BPGraph are only available for **SPSD**, **SSSD** and **kNN** on selected datasets.

Environment. All experiments were carried out on a server featuring an AMD EPYC 7643 Processor with a 256 MB L3 cache size and 256GB of memory. We compiled all implementations using G++ 9.4.0, enabling the O3 optimization and OpenMP support. Each system was configured to utilize 16 threads by default.

7.2 Overall Performance

We compare the overall performance of all systems and the results are reported in Table 3. We also report the speedup achieved by uBlade against the next best system for all compared scenarios.

We first discuss BPGraph since its performance numbers are only available for selected scenarios. The results show that BPGraph can outperform Sage on kron, soc, and uk for the **SSSD** and **SPSD** queries. This advantage may be attributed to BPGraph's precomputation of all paths from source to target, which enhances its online sampling process. However, it is worth pointing out that such precomputation cannot be applied to the **kNN** query due to the exponential number of possible paths from the source to all nodes in the graph. Conversely, Sage consistently outperforms BPGraph for smaller datasets such as hep and gnu. This difference in performance is attributed to Sage utilizing collective gathering techniques [56] to reduce computational overhead. However, the technique is known to be less effective on larger datasets with more distinct distance values.

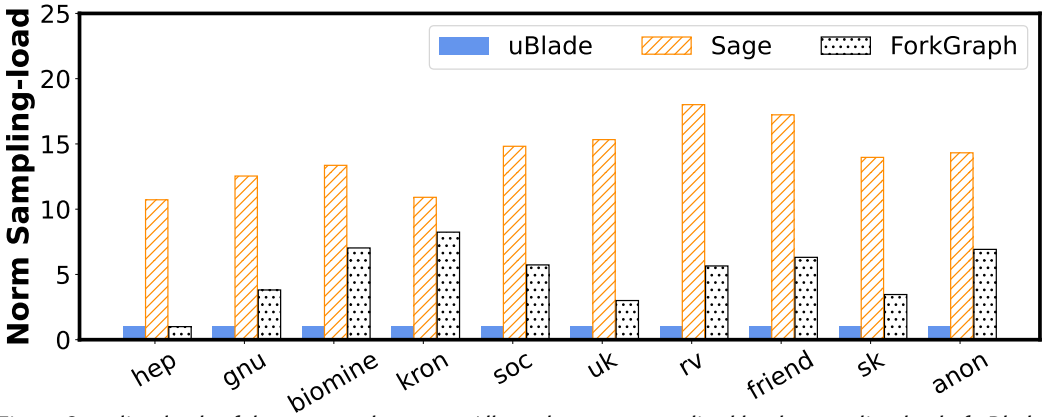
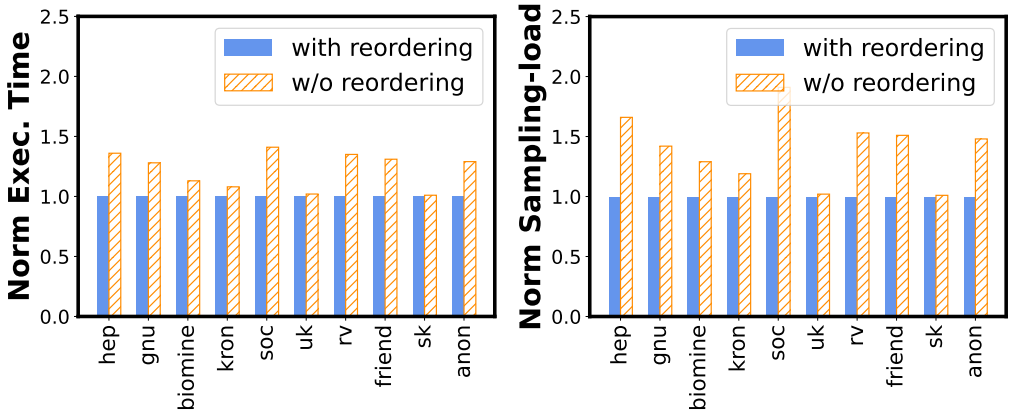


Fig. 7. Sampling-loads of the compared systems. All numbers are normalized by the sampling-load of uBlade.

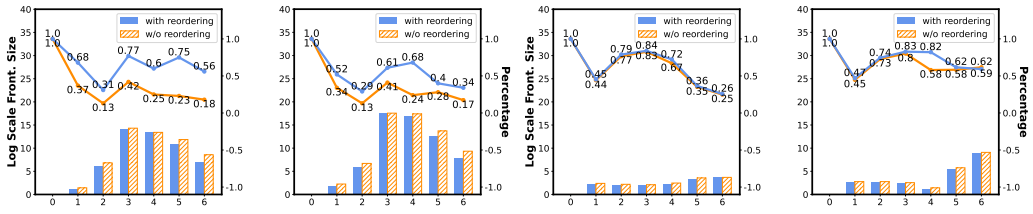


(a) Normalized execution time (b) Normalized sampling-load

Fig. 8. Performance comparison with and w/o reordering.

Next, we observe that ForkGraph consistently outperforms Sage for all scenarios. This result is surprising since Sage is tailored for uncertain graph processing whereas ForkGraph is a more generic graph processing system. Upon closer inspection of Sage's implementation, we discovered that it samples all possible worlds of active nodes, even when these nodes may not be reachable from the source node in all possible worlds. Hence, Sage is more adversely affected by the overhead of random number generations than ForkGraph. Thanks to ForkGraph's flexible design, we can adapt Deterministic Sampling on ForkGraph for uncertain graph processing while avoiding sampling nodes on possible worlds where they remain unvisited. We confirm this observation by profiling their respective sampling-load in Figure 7 (for the **SSSD** query). On average, Sage incurs 2 times more sampling-load than that of ForkGraph.

Finally, uBlade achieves 1-2 orders of magnitude speedups compared with Sage and BPGraph for all the datasets. Further, uBlade also significantly outperforms ForkGraph as evidenced by the speedup numbers reported in Table 3 as well as the sampling-load profiling results in Figure 7. In particular, uBlade saves at least 80% sampling-load compared with ForkGraph and at least 90% sampling-load compared with Sage. However, there is one exception observed in the case of the hep dataset, where ForkGraph outperforms uBlade. This anomaly can be attributed to the reordering technique employed by uBlade, which necessitates running the first two iterations *twice*. This



(a) Varying iters on rv (b) Varying iters on anon (c) Varying iters on uk (d) Varying iters on sk

Fig. 9. Frontier size and share ratio comparison with and w/o reordering. Frontier sizes are in log scale.

effectively doubles the sampling-load for uBlade on the hep dataset, as complete traversals on hep often terminate within 2-3 iterations.

The speedup values are notably higher for **SPSD** compared to the other queries. This is because we enable a query-specific optimization, where nodes that are either unreachable from the source node or unable to reach the target node are pruned before any sampling occurs. As a result, there are significantly fewer nodes to process during traversal, leading to a substantially reduced sampling-load. We applied this optimization to ForkGraph as well. However, the improvement in ForkGraph is minimal. This is due to the fact that ForkGraph processes partitions sequentially, incurring additional processing overhead for each partition, even if it contains only one active node.

To summarize, uBlade demonstrates significant speedup gains when compared to state-of-the-art systems designed for uncertain graph processing. Given the similarity in performance characteristics across the compared approaches for different query types, we have selected the **SSSD** query as a focal point for conducting a more detailed analysis of our uBlade system.

7.3 Reorder Performance

Figure 8 provides a comparison of uBlade's performance with and without the use of reordering. For most datasets, reordering enhances execution with a speedup of 22.4% on average by reducing the sampling-load by 28.7%. Note that the speedup in runtime is slightly less than the reduction ratio in sampling-load. This is because we count generated random numbers as the sampling-load. Reordering primarily reduces the total number of random number generations, but it does lead to a slight increase in other computational tasks, such as distance updates. This explains the minor reduction in time improvement when compared to the reduction in sampling-load.

We also conducted profiling of the frontier size (in log scale) and the share ratio for scenarios where reordering proved either effective or ineffective, as depicted in Figure 9.

The *frontier size* is the number of active nodes in each iteration, which we use to analyze the sampling-load of each iteration. The *share ratio* for each iteration is computed by summing the number of associated possible worlds for each active node and then dividing it by the product of the number of distinct possible worlds with at least one active node and the number of distinct active nodes in that iteration. In essence, when all active nodes share the same set of associated possible worlds, the share ratio is equal to 1.

Examining Figure 9a and 9b, the initial two iterations involve very few nodes, with the majority of nodes becoming active by the third iteration. This suggests that the first two steps significantly influence which nodes a possible world might visit. By calculating the number of nodes visited during these initial two steps, we can estimate the similarity between two possible worlds and improve the share ratio effectively with reordering. This observation also explains why reordering does not yield favorable results for the uk and sk datasets. Additionally, Figure 9c and 9d demonstrate that the frontier size for these two datasets experiences gradual growth. Unlike social networks,

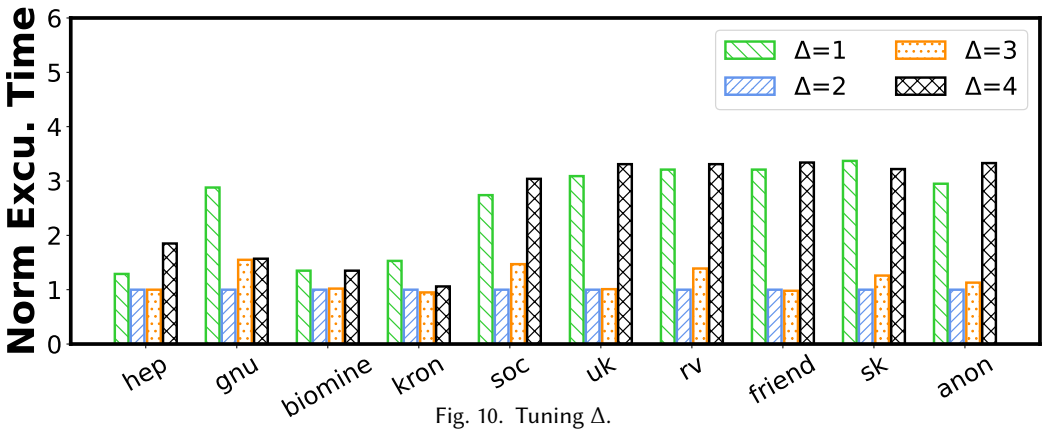


Fig. 10. Tuning Δ.

there is no explosive expansion in the frontier size at the third iteration; instead, it steadily increases with each step as the traversal progresses. Given this observation, the performance of the first two iterations cannot reliably predict the nodes most visited by the traversal, which accounts for the reordering's ineffectiveness on these datasets.

Moreover, the results indicate that in the early iterations, the frontier size constitutes only a negligible fraction of the total frontier size across all iterations. Consequently, even if the reordering optimization proves ineffective, the associated overhead remains minimal and exerts little influence on the overall performance.

7.4 Parameter Tuning

Varying Δ. Figure 10 provides the average performance of uBlade's across various datasets while varying the parameter Δ. Our observations are as follows: First, when Δ is set to a large value, uBlade behaves similarly to Bellman-Ford by expanding all neighbors of an active node. This leads to additional computational sampling-load because different possible worlds are required to visit the same node at the same hop to share computations. Second, when Δ is set to a small value, the traversal closely resembles Dijkstra's algorithm. However, this also results in increased sampling-load since different possible worlds must visit nodes with the same shortest distance. Hence, there exists a favorable range for selecting Δ, with $\Delta = 2$ emerging as the optimal choice in our experiments. Ideally, adaptive selection of Δ according to different traversal patterns would be beneficial. We leave it as a future work.

Varying batch size B**.** Figure 11 offers insight into the average performance of uBlade across various datasets while adjusting the batch size, denoted as B. When the batch size B is set to a small value, the possible worlds undergo more traversals, leading to suboptimal locality and less opportunity for sharing computations between possible worlds. Conversely, when B is set to a large value, the additional sampling-load incurred by processing larger batches outweighs the benefits of computation sharing, resulting in a more resource-intensive process. Notably, on datasets sk and uk, due to their distinctive graph structures, possible worlds seldom have the opportunity for shared computation. Consequently, as B increases, their performance declines. Taking all datasets into account, a batch size of $B = 20$ yields the best performance empirically.

Varying thread numbers. In Figure 12, we conduct scalability tests for ForkGraph and uBlade with varying numbers of threads on two large datasets: `friend` and anon. Here, the node s with the highest degree is selected, and **SPSD** is conducted from s with 100 samples.

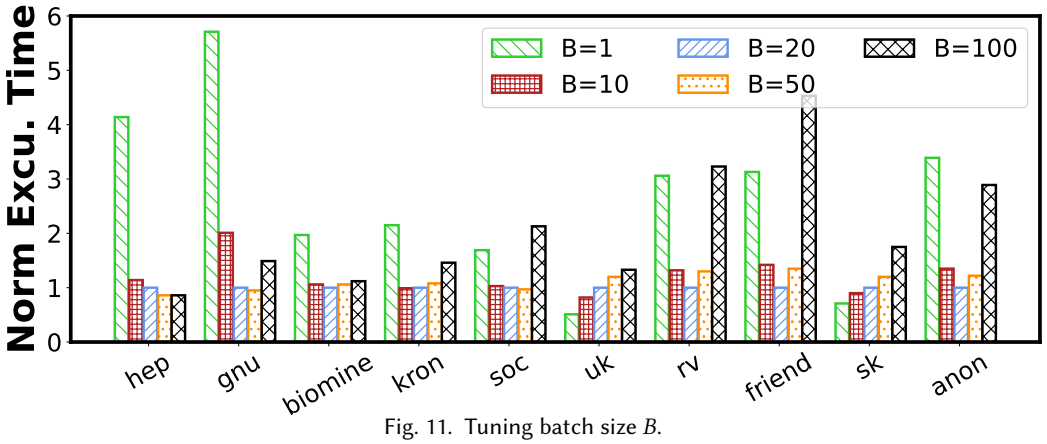
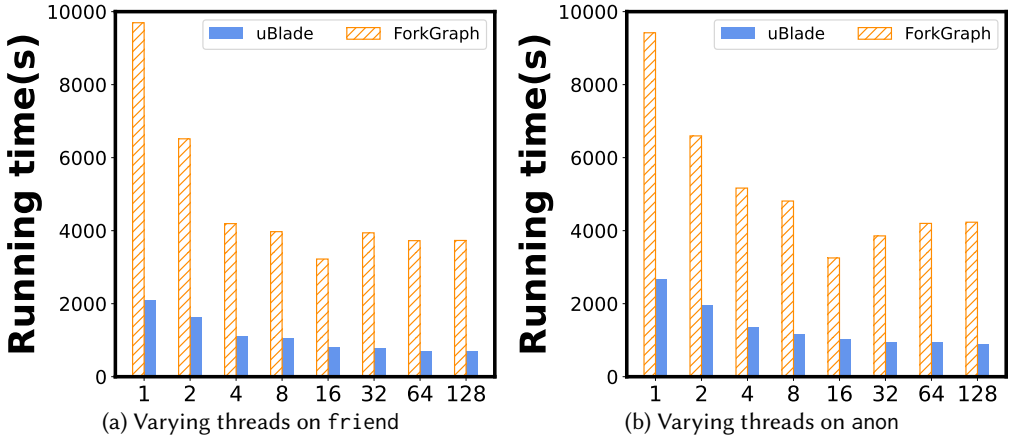


Fig. 11. Tuning batch size B.



(a) Varying threads on `friend` (b) Varying threads on anon

Fig. 12. Tuning thread numbers.

The results show that both uBlade and ForkGraph scale well up to 16 threads. While uBlade shows marginal performance gains with even more threads, ForkGraph does not show similar improvements beyond 16 threads. This is because ForkGraph assigns one thread to each possible world, and when the thread number exceeds the batch size, there are contention issues in processing the same graph partition. In contrast, uBlade assigns threads to jointly process the frontier, showing better scalability. Nevertheless, the potential for scalability enhancement remains, primarily hindered by random access patterns causing I/O bottlenecks, especially in uncertainty graph query processing. We leave it as future work.

8 RELATED WORKS

Uncertain Graph Queries. Numerous uncertain graph analyses have been presented in past literature. Key areas of focus include connectivity [15], kNN [42], centrality [44], influence maximization [13], and clustering [27]. Notably, reliability queries [24] and shortest distance queries [54] form the foundation of these analyses. Addressing the challenges of processing an enormous number of possible worlds, two primary approaches emerge: sampling-based and index-based. The

former samples possible worlds for query resolution, leveraging methods like Monte-Carlo [14], Recursive [19], and Recursive Stratified sampling [32]. The latter approach builds an offline index on the uncertain graph to bolster query efficiency, utilizing techniques such as BFS Sharing [55] and Probabilistic Trees [38]. We refer to the survey [22] for an in-depth discussion.

Uncertain Graph Systems. Several systems are tailored to support uncertain graph query processing. Zou et al. [56] pioneered this realm with a system employing *collective gathering* to reduce redundant computations. Yet, its effectiveness diminishes in scenarios with more distinct values per node [29]. The GPU-optimized BPGraph[51] system precomputes all potential paths for queries, focusing on fine-grained parallelism for processing unique edges in chosen paths. However, its approach can exponentially increase precomputed paths and is only suited for s-t queries. The state-of-the-art uncertain graph system, Sage[29], employs Deterministic Sampling to circumvent possible world materialization. We introduce uBlade as a solution to address the limitations of existing systems, which struggle with inefficient graph traversal, high random number generation costs, and the added sampling-load associated with batch processing.

Concurrent Graph Processing Systems. There is a rapidly growing area of research in developing scalable systems for concurrent graph processing systems [18, 36, 37, 46, 47, 50, 52, 53]. These systems capitalize on multiple queries being executed on the same graph and exploit the sharing opportunities. Joint frontier execution among queries is a key technique to enable sampling-load sharing [18, 36, 47]. Furthermore, the graphs are divided into partitions which are processed in order to minimize I/O cost [37, 52, 53]. While uBlade is inspired by this research trajectory, it uniquely introduces optimizations such as batch traversal with Δ-stepping and reordering, both novel contributions to the domain.

9 CONCLUSION

In this study, we presented uBlade, a state-of-the-art system tailored for efficient batch processing of uncertain graph queries. At its core, uBlade integrates batch processing using Δ-stepping for work-efficient parallelism. The introduction of Quasi-Sampling further curtails the computational overhead associated with random number generation during traversals across possible worlds. Additionally, our unique reordering optimization adeptly diminishes the supplemental sampling-load inherent in the batch processing framework. Through rigorous evaluations, it becomes evident that uBlade outperforms leading benchmarks, delivering a performance boost of $1-2$ orders of magnitude in uncertain graph processing.

Future Work. Our approach can handle the situation where the original graph can be stored in memory. This is because we do not materialize the active edges in the possible worlds with Quasi-sampling. Conversely, when the original uncertain graph exceeds the memory, our system currently lacks support for external memory processing. However, since our system aligns with the vertex-centric processing framework, this compatibility opens avenues for leveraging numerous existing systems and methods that optimize disk-based vertex-centric processing [5, 49].

Acknowledgement. This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002), and the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No: MOE2019-T2-2-065).

REFERENCES

[1] E. Adar and C. Re. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 30(2):15–22, 2007.
[2] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.
[3] P. Boldi, F. Bonchi, A. Gionis, and T. Tassa. Injecting uncertainty in graphs for identity obfuscation. *Proceedings of the VLDB Endowment*, 5(11):1376–1387, 2012.

[4] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208, 2009.

[5] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He. Venus: Vertex-centric streamlined graph computation on a single pc. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1131–1142. IEEE, 2015.

[6] R. Cheng, L. Chen, J. Chen, and X. Xie. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 672–683, 2009.

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

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

[9] Y. Chi, L. Guo, and J. Cong. Accelerating sssp for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–200, 2022.

[10] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[11] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304, 2017.

[12] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[13] S. Eshghi, S. Maghsudi, V. Restocchi, S. Stein, and L. Tassiulas. Efficient influence maximization under network uncertainty. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 365–371. IEEE, 2019.

[14] G. S. Fishman. A comparison of four monte carlo methods for estimating the probability of st connectedness. *IEEE Transactions on reliability*, 35(2):145–155, 1986.

[15] X. Gao and Y. Gao. Connectedness index of uncertain graph. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 21(1):127–137, 2013.

[16] J. Ghosh, H. Q. Ngo, S. Yoon, and C. Qiao. On a routing problem within probabilistic graphs and its application to intermittently connected networks. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 1721–1729. IEEE, 2007.

[17] A. Goyal, W. Lu, and L. V. Lakshmanan. Celf++ optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*, pages 47–48, 2011.

[18] M. Hauck, M. Paradies, and H. Fröning. Can modern graph processing engines run concurrent queries efficiently? In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–6, 2017.

[19] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *Proceedings of the VLDB Endowment*, 4(9):551–562, 2011.

[20] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[21] V. Kassiano, A. Gounaris, A. N. Papadopoulos, and K. Tsichlas. Mining uncertain graphs: An overview. In *International Workshop of Algorithmic Aspects of Cloud Computing*, pages 87–116. Springer, 2016.

[22] X. Ke, A. Khan, and L. L. H. Quan. An in-depth comparison of s-t reliability algorithms over uncertain graphs. *Proceedings of the VLDB Endowment*, 12(8):864–876, 2019.

[23] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146, 2003.

[24] A. Khan, F. Bonchi, A. Gionis, and F. Gullo. Fast reliability search in uncertain graphs. In *Proceedings of the 17th International Conference on Extending Database Technology*, pages 535–546, 2014.

[25] A. Khan and L. Chen. On uncertain graphs modeling and queries. *Proceedings of the VLDB Endowment*, 8(12):2042–2043, 2015.

[26] A. Khan, Y. Ye, and L. Chen. On uncertain graphs. synthesis lectures on data management. *Morgan & Claypool*, 2018.

[27] G. Kollios, M. Potamias, and E. Terzi. Clustering large probabilistic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 25(2):325–336, 2013.

[28] N. J. Krogan, G. Cagney, H. Yu, G. Zhong, X. Guo, A. Ignatchenko, J. Li, S. Pu, N. Datta, A. P. Tikuisis, et al. Global landscape of protein complexes in the yeast saccharomyces cerevisiae. *Nature*, 440(7084):637–643, 2006.

[29] E. Lee, S. H. Noh, and J. Seo. Sage: A system for uncertain network analysis. *Proceedings of the VLDB Endowment*, 15(13):3897–3910, 2022.

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

[31] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *The VLDB Journal*, 20:249–275, 2011.
[32] R.-H. Li, J. X. Yu, R. Mao, and T. Jin. Recursive stratified sampling: A new framework for query evaluation on uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering*, 28(2):468–482, 2015.
[33] Y. Li, J. Fan, G. Ovchinnikov, and P. Karras. Maximizing multifaceted network influence. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering*, pages 446–457. IEEE, 2019.
[34] Y. Li, J. Fan, Y. Wang, and K.-L. Tan. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 30(10):1852–1872, 2018.
[35] Y. Li, D. Zhang, and K.-L. Tan. Real-time targeted influence maximization for online advertisements. *Proceedings of the VLDB Endowment*, 8(10), 2015.
[36] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416, 2016.
[37] S. Lu, S. Sun, J. Paul, Y. Li, and B. He. Cache-efficient fork-processing patterns on large graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1208–1221, 2021.
[38] S. Maniu, R. Cheng, and P. Senellart. An indexing framework for queries on probabilistic graphs. *ACM Transactions on Database Systems (TODS)*, 42(2):1–34, 2017.
[39] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
[40] P. Parchas, F. Gullo, D. Papadias, and F. Bonchi. 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*, pages 967–978, 2014.
[41] V. Podpečan, Ž. Ramšak, K. Gruden, H. Toivonen, and N. Lavrač. Interactive exploration of heterogeneous biological networks with biomine explorer. *Bioinformatics*, 06 2019.
[42] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. K-nearest neighbors in uncertain graphs. *Proceedings of the VLDB Endowment*, 3(1-2):997–1008, 2010.
[43] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[44] A. Saha, R. Brokkelkamp, Y. Velaj, A. Khan, and F. Bonchi. Shortest paths and centrality in uncertain networks. *Proceedings of the VLDB Endowment*, 14(7):1188–1201, 2021.
[45] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.
[46] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li. Thunderrw: an in-memory graph random walk engine. *Proceedings of the VLDB Endowment*, 14(11):1992–2005, 2021.
[47] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4):449–460, 2014.
[48] L. G. Valiant. The complexity of enumeration and reliability problems. *siam Journal on Computing*, 8(3):410–421, 1979.
[49] X. Wang, D. Wen, L. Qin, L. Chang, Y. Zhang, and W. Zhang. Scaleg: A distributed disk-based system for vertex-centric graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 35(2):2019–2033, 2023.
[50] C. Ye, Y. Li, S. Sun, and W. Guo. gsword: Gpu-accelerated sampling for subgraph counting. *Proceedings of the ACM on Management of Data*, 2(1):1–26, 2024.
[51] H. Zhang, L. Li, H. Liu, D. Zhuang, R. Liu, C. Huan, S. Song, D. Tao, Y. Liu, C. He, et al. Bring orders into uncertainty: enabling efficient uncertain graph processing via novel path sampling on multi-accelerator systems. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–14, 2022.
[52] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu. {CGraph}: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452, 2018.
[53] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen. Graphm: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.
[54] J. Zhou, F. Yang, and K. Wang. An inverse shortest path problem on an uncertain graph. *Journal of Networks*, 9(9):2353, 2014.
[55] R. Zhu, Z. Zou, and J. Li. Top-k reliability search on uncertain graphs. In *2015 IEEE International Conference on Data Mining*, pages 659–668. IEEE, 2015.
[56] Z. Zou, F. Li, J. Li, and Y. Li. Scalable processing of massive uncertain graph data: A simultaneous processing approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 183–186. IEEE, 2017.

Received October 2023; revised January 2024; accepted February 2024