ShareDP: Finding k Disjoint Paths for Multiple Vertex Pairs

Zhiqiu Yuan[1], Youhuan Li[2], Lei Zou ✉[1], and Linglin Yang[1]

[1] Peking University
{yuanzhiqiu, linglinyang}@stu.pku.edu.cn, zoulei@pku.edu.cn
[2] Hunan University
liyouhuan@hnu.edu.cn

Abstract. Finding k disjoint paths (kDP) is a fundamental problem in graph analysis. For vertices s and t, paths from s to t are said to be *disjoint* if any two of them share no common vertex except s and t. In practice, disjoint paths are widely applied in network routing and transportation. In these scenarios, *multiple* kDP queries are often issued simultaneously, necessitating efficient batch processing. This motivates the study of *batch* kDP query processing (batch-kDP). A straightforward approach to batch-kDP extends batch simple-path enumeration with disjointness checks. But this suffers from factorial computational complexity. An alternative approach leverages single-query algorithms that avoid this by replacing the graph with a converted version. However, handling each query independently misses opportunities for shared computation. To overcome these limitations, we propose ShareDP, an algorithm for batch-kDP that shares the computation and storage across kDPs. ShareDP merges converted graphs into a shared structure, then shares the traversals and operations from different queries within this structure. Extensive experiments on 12 real-world datasets confirm the superiority of ShareDP over comparative approaches.

Keywords: disjoint paths, graph analysis, batch query processing

1 Introduction

Finding k disjoint paths (kDP) is a fundamental challenge in graph analysis [4, 13, 5, 8, 2, 11]. Given a source and a target vertex, kDP identifies k paths that do not share any vertices except for the source and target[3]. Fig. 1a illustrates an example with disjoint paths p_1 (red) and p_2 (blue). Applications of kDP include cyber-security, network fault tolerance, and load balancing [6, 9, 8, 2, 11].

In practice, multiple kDP queries are often processed in batches. For example, as communication networks scale with the Internet, numerous routing queries are generated quickly, necessitating high-throughput processing. This paper focuses

[3] We focus on vertex-disjoint paths, as edge-disjoint path finding problem can be reduced to vertex-disjoint version in polynomial time [12].

on the problem of batch kDP query processing (`batch-`k`DP`), where given a graph G and a set of kDP queries Q, returns k disjoint paths for each query in Q.

Existing studies on kDP can be categorized into *single-query* [4, 13, 5, 8, 2, 11] and *single-source* [6, 3, 1]. (1) *Single-query* methods include *flow-augmenting path-based* [4, 13, 5] and *dissimilar path-based* [8, 2, 11]. The latter extends simple path enumeration by incorporating disjointness constraints, but suffers from factorial time complexity (see Sec. 3.1). In contrast, the former achieves linear time complexity using *split-graphs* [4, 5, 13], where finding disjoint paths reduces to identifying flow-augmenting paths in the split-graphs. (2) *Single-source* kDP seeks disjoint paths from a source to all other vertices, with current studies primarily theoretical and limited to $k \leq 4$ [1]. (3) To our knowledge, this paper is the first to address `batch-`k`DP` with general k.

For `batch-`k`DP`, we can adapt methods from single-query kDP: (1) *Dissimilar path-based* methods can extend batch path enumeration algorithms [15] but also face factorial complexity. (2) *Flow-augmenting path-based* methods handle each query in linear time but require independent split-graphs for each query, missing opportunities for shared computation

This leads to the question: can we apply existing batch-processing techniques from other problems? Unfortunately, current techniques are designed for queries in the same graph [15, 10, 7, 14], while our problem involves different kDPs executed over different graphs (split-graphs). The first step is to unify the split-graphs, which is not covered in the literature. Thus, we propose `ShareDP`, a batch-kDP algorithm that aims to `share` common computations and storage across a batch of kDP queries. The overall framework of `ShareDP` is illustrated in Fig. 2 (explained in Sec. 5.1).

`ShareDP` incorporates the following key strategies: (1) We consolidate individual split-graphs into a unified structure, represented implicitly via result sets, simplifying construction and updates. (2) Using the merged split-graph, we enable concurrent kDP query processing by consolidating traversals and operations leveraging tagged data structures, reducing redundancy and improving efficiency. In summary, our contributions are:

- *Merged Split-Graph Representation.* The `ShareDP` framework introduces a novel merged split-graph representation that consolidates individual split-graphs into a unified structure, enabling dynamic sharing of traversals.
- *Optimized Path-Finding with Shared Traversals.* The framework combines traversals across multiple queries, consolidating common operations into single steps, reducing redundancy and improving efficiency.
- *Proven Efficiency and Scalability.* Extensive evaluations on 12 real-world datasets demonstrate that `ShareDP` consistently achieves the lowest runtime across various k settings, confirming its efficiency and scalability.

2 Preliminaries

We use p for a path and P for a set of paths. A vertex v is an *intermediate* vertex in path p if it is neither the starting nor the ending vertex. If v is intermediate in any path in P, it is called P-*inner*. A path p is simple if no vertex appears more than once. $V(P)$ and $E(P)$ refer to the set of vertices and edges that appear in at least one path in P. While the focus here is on directed graphs, the method can be easily adapted to undirected graphs.

Disjoint Paths Given a graph $G = (V, E)$ and two simple paths p_1 and p_2 from s to t, p_1 and p_2 are *disjoint* if they share no common vertex except s or t.

Problem Statement Given a graph $G = (V, E)$, a parameter k, and a set of kDP queries $Q = \{q_0, q_1, ..., q_\omega\}$ where each query q_i is a vertex pair (s_i, t_i) $(i = 1, ..., \omega)$, `batch-`k`DP` is to find k disjoint paths for each query $(s_i, t_i) \in Q$.

3 Related Work

3.1 kDP problems

Single-query. Approaches to single-query kDP can be divided into *flow-augmenting path-based methods* [4, 13, 5] and *dissimilar path-based methods* [8, 2, 11]. Flow-augmenting methods use *split-graphs* [4, 5], achieving linear time complexity (Sec. 4). Dissimilar path-based methods, which define dissimilarity as disjointness, include *penalty-based*, *dissimilarity-based*, and *plateaus-based* methods [8, 11]. Plateaus-based methods, however, may fail to find solutions, as they depend on shared branches between two shortest spanning trees, which may not present in all kDP solutions. Thus, we exclude them from comparison.

Penalty-based and dissimilarity-based methods reduce to path enumeration with disjointness constraints, either by *(1)* marking vertices in previous paths as inaccessible or by *(2)* verifying disjointness before adding new paths. However, both lead to factorial time complexity in the worst case. For instance, if a path p is not part of any solution, all computations involving p in the result set will fail to yield a solution. Consequently, alternative path orderings must be attempted (e.g., adding p_1 first, then p_2, and so on). In the worst case, every possible path ordering must be evaluated, leading to factorial time complexity. We may also *(3)* first identify all paths and then select a subset of k disjoint paths, but this also faces the path-ordering challenge, resulting in the same factorial complexity. Our experiments (Sec. 6.2) show these methods time out on large graphs.

k shortest dissimilar path finding [2] can solve kDP but is slower due to path enumeration in ascending length order, as noted in [11]. Thus, we exclude them from comparison.

Single-source. The problem of finding k independent spanning trees [6, 3, 1] is related to kDP, where paths in different spanning trees are disjoint. While this provides solutions for vertex pairs formed by the root and other vertices, the problem remains an open challenge for $k > 4$. For $k \leq 4$, this method was over 10 times slower than our approach in our experiments (Sec. 6.2).

To our knowledge, no prior work addresses the `batch-`k`DP` problem.

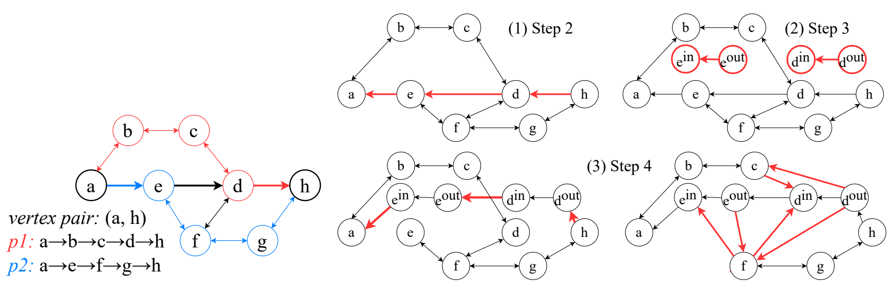
3.2 Batch Query Processing

Batch query processing has been studied for other graph problems [15, 10, 7, 14], focusing on queries within the same graph. [15] explores batch simple path enumeration, leveraging shared computations and caching results. [10] investigates batch subgraph isomorphism with a unified join plan. [7] implements caching and query decomposition for shortest path queries. Batch BFS queries [14] exploit overlaps among frontiers across concurrent BFS runs. In contrast, our problem involves different kDP queries on different graphs (split-graphs), requiring unification of these graphs, which remains unexplored in the literature.

4 Baseline

The baseline method for batch-kDP solves each query using flow-augmenting path-based methods, which rely on the concept of *split-graphs* [4, 5, 13].

Definition: Split-Graph [13] Given a graph $G = (V, E)$ and a set P of disjoint paths from s to t, the split-graph $\mathcal{G}_{G,P} = (\mathcal{V}_{G,P}, \mathcal{E}_{G,P})$ is constructed as follows: (1) Initializing $\mathcal{V}_{G,P} = V$ and $\mathcal{E}_{G,P} = E$. (2) For each edge in $E(P)$, reversing the corresponding edge in $\mathcal{E}_{G,P}$. (3) Splitting vertices $v \in V(P) \setminus \{s, t\}$ into v^{in} and v^{out}, and connecting them accordingly. (4) Replacing edges in $\mathcal{E}_{G,P}$ with updated vertex connections, preserving incoming and outgoing edges.



(a) Disjoint paths for (a, h). (b) Split-graph with $P = \{p_1 = \{\text{a,e,d,h}\}\}$.

Fig. 1: Examples of disjoint paths and split-graph.

Given a graph G and vertices s and t, the algorithm proceeds as follows: (1) Initialize $P = \emptyset$ and $\mathcal{G}_{G,P} = G$. (2) Find the first path p_1 in $\mathcal{G}_{G,P}$ using any path-finding algorithm (e.g., BFS), forming $P_1 = \{p_1\}$, and update $\mathcal{G}_{G,P}$ to $\mathcal{G}_{G,P_1}$. (3) Search for p_2 in $\mathcal{G}_{G,P_1}$, yielding $P_2 = \{p_1, p_2\}$, and adjust P_2 following an approach similar to augmenting flows [4]. Then update $\mathcal{G}_{G,P_1}$ to $\mathcal{G}_{G,P_2}$. (4) Search for p_3 in $\mathcal{G}_{G,P_2}$. More paths are found in a similar manner.

5 ShareDP

Given a set of kDP queries, the key improvement over the baseline in our approach lies in sharing computations across queries. We demonstrate this with the indochina-2004 dataset (statistics in Tab. 1). In both the first and last iterations (i.e., finding the first and k-th disjoint paths), exploration of over 60% of the vertices are shared across levels, and in more than half of the levels, over 80% are shared. This illustrates the potential for shared exploration in the batch-kDP problem. Based on this, we propose the `ShareDP` algorithm.

5.1 Algorithm Overview

The `ShareDP` framework (Fig. 2) takes as input a graph G, a set of kDP queries Q, and an integer k. In each iteration, the algorithm identifies one path for each query. In the i-th iteration, the algorithm operates on a merged split-graph G' , conceptually the union of individual split-graphs for each query (top half of Fig. 2(a)). G' is represented implicitly using the current result sets of all queries (bottom half of Fig. 2(a)). During path-finding, traversals are combined across queries, conceptually visualized as collapsing multiple planes where the traversal for each query flows into a unified plane (top half of Fig. 2(b)). Common operations are consolidated into a merged step by tagging data structures with query sets (bottom half of Fig. 2(b)). Paths identified in each iteration update the result sets, which in turn update G' for the next iteration (bottom of Fig. 2(b)).

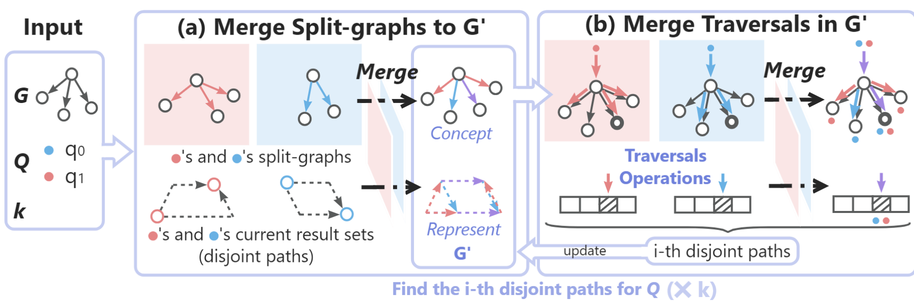


Fig. 2: Framework of `ShareDP`. Colored balls (e.g., the blue ball for query q_0) represent queries. Each iteration identifies a path for each query by conducting a combined search on a merged split-graph G'. See Sec. 5.1 for details.

5.2 Algorithm Details

Merge the Split-Graphs. We merge the split-graphs from the perspective of a vertex v, focusing on its out-neighbors (in-neighbors analysis is symmetric). They are derived from the original graph's out-neighbors, reversed edges, and vertex splitting (Sec. 4). To manage edge reversals and vertex positions, we define $nexthops$, $prehops$, $isPinner$, isS, and isT, which also represent the current result sets $\{P\}$. Specifically, $prehops_{u,v}$ represents queries where v is u's $prehop$; $isPinner_v$, isS_v, and isT_v represent the set of kDP queries where v is P-inner, s, and t, respectively. With these definitions, given a query set B and a vertex

Algorithm 1: GetOutNeighbors

```
Input: Vertex v, set of kDPs B, original graph G, {P}
Output: {(u, B')} // u: a neighbor of v, B': corresponding kDP subset.
S = ∅
if v is new_in // v is v_in. then
    foreach u in prehops_v // Reversed edges: v's prehops. do
        Add (u, B ∩ prehops_{v,u}) to S
else // v is v_out or v.
    foreach u in out-neighbors of G do
        haveset = B \ nexthops_{v,u} // Excluding reversed edges.
        Add (u^in, haveset ∩ isPinner_u) to S // Edges to P-inner
          vertices are redirected to new^in.
        Add (u, haveset \ isPinner_u) to S // Otherwise no redirecting.
    Add (v^in, B ∩ isPinner_v) to S // The edge from v^out to v^in
return S
```

v, the procedure for acquiring v's out-neighbors is shown in Alg. 1. The merging process uses the result sets $\{P\}$ from each query $q \in B$ to determine neighbors in the merged split-graph G', simplifying neighbor retrieval and improving efficiency compared to constructing a supergraph (Sec. 6.2).

Combine Traversals. The ShareDP algorithm combine the traversals by tagging data structures with query sets. While any path-finding algorithm could be used, we employ bidirectional BFS as an example here. When a frontier vertex v expands in the forward search (Alg. 2, backward search proceeds similarly), its kDP set is divided into subsets, and each neighbor is processed only once for the kDPs in that subset. If frontiers share the same neighbor, their kDP sets are merged, enabling continued shared traversal. The complete algorithm of `ShareDP` is provided in the appendix [16].

Algorithm 2: ForwardExpandFrontier

```
Input: Frontier (vertex v, queries B), original graph G, {P}
InOut: Data structures for searching and path recording.
B = B \ undone // Skip queries that have already found the ith path.
foreach (u, B') ∈ GetOutNeighbors(v, B, G, {P}) // See Alg. 1. do
    D = B' \ s-seen_u // Exclude queries that have already visited u.
    s-seen_u ∪= D; pred_{u,v} ∪= D // Mark visited and record predecessor.
    meet = D ∩ t-seen_u // The forward and backward searches meet.
    joint_u ∪= meet; undone \= meet // Mark queries as completed.
    s-nextqueue_u ∪= (D \ meet) // Add to the queue.
```

6 Experimental Evaluation

We implemented all algorithms in C++ and tested on a Ubuntu machine with 512 GB RAM and Intel(R) Xeon(R) Platinum 8352V 2.10GHz CPU.

6.1 Experimental Setup

Algorithms. 1) *ShareDP*: Our approach using bidirectional BFS (Sec. 5.2). 2) *ShareDP-*: ShareDP with supergraph representation. 3) *maxflow* [13]: Baseline method (Sec. 4). 4) *BatchEnum* [15]: State-of-the-art path enumeration with adaptations (3) for our problem (Sec. 3.1).[4] 5) *SCB+* [11]: State-of-the-art dissimilar path finding method (adaptation (2)). 6) *Penalty* [8]: Another dissimilar path finding method (adaptation (1)). 7) *IST* [6]: Single-source method for $k = 2$ (case for $k > 4$ is open challenge, see Sec. 3.1).

Datasets and Queries. We evaluate on 12 real-world datasets (Tab. 1), sourced from SNAP, NetworkRepository, and LAW. These include the datasets used in [11] (first 6 rows), supplemented by 6 larger datasets for improved comparison (last 6 rows). For each graph, we generate 1000 vertex pairs with kDP solutions, starting from k=50 down to k=2. Candidate pairs are selected based on vertex degree $\geq k$. If fewer than 20% succeed, k is reduced. The maximum k is termed k_{max}. Algorithms are evaluated on 4–5 ks per graph, based on k_{max}.

6.2 Experimental Result

Comparing Algorithms when varying *k* We evaluated different algorithms across various k values (Fig. 3). The y-axis shows average runtime per query (seconds), and the x-axis represents k. Queries exceeding 200 seconds were terminated and recorded as such.

ShareDP (brown) consistently outperformed others with the lowest runtime. Most dissimilarity-based methods failed to complete for nearly all ks, When they did run, they were more than 10 times slower than flow-based methods (maxflow and ShareDP). IST was at least 10 times slower for the only k it handled (k=2).

ShareDP's advantage over maxflow increased as k grew. For graphs with fewer disjoint paths (e.g., id and uk), ShareDP significantly outperformed maxflow. For graphs with more disjoint paths (e.g., sk and tw), the advantage of ShareDP was more pronounced at larger k, as finding a small number of disjoint paths in these graphs is easy, reducing the relative advantage of ShareDP in simpler cases. For similar reasons, ShareDP showed a greater advantage on larger graphs (e.g., the last row). These results highlight the efficiency and scalability of ShareDP.

Effect of the number of *k*DPs We analyzed performance as the number of kDPs ($|Q|$) varied from 1 to 1000 with k=10 across all datasets (Fig. 4). The

[4] BatchEnum reuses path enumeration results across queries, requiring complete enumeration, which prevents adaptations (1) and (2).

Table 1: Properties and evaluated k values of datasets

Name	**Dataset**	$\|V\|$	$\|E\|$	**Type**	**D**	k_{max}	k
rt	reactome	6.3K	147K	Biology	24	50	2,10,15,20,50
am	amazon	334K	925K	Web	44	15	2,5,8,10,15
ts	twitter-social	465K	834K	social	8	50	2,10,15,20,50
bs	berkstan	685K	7M	Web	208	10	2,5,8,10
wg	web-google	875K	5M	Web	24	10	2,5,8,10
sk	skitter	1.6M	11M	infrastructure	31	50	2,10,15,20,50
fl	flickr-links	1.7M	16M	Social	14	50	2,10,15,20,50
fg	flickr-growth	2.3M	33.1M	Social	13	50	2,10,15,20,50
id	indochina-2004	7.4M	194M	Web	25	10	2,5,8,10
ar	arabic-2005	22.7M	640M	Web	29	20	2,5,10,15,20
uk	uk-2005	39.5M	1.9B	Web	22	10	2,5,8,10
tw	twitter-2010	42M	1.5B	Social	16	50	2,10,15,20,50

y-axis shows average runtime per query (seconds), and the x-axis shows $|Q|$. As shown in Fig. 4, `ShareDP`'s runtime decreases as $|Q|$ increases, outperforming `maxflow` at small $|Q|$. This is due to shared computation when more kDPs are processed together. The improvement slows with larger $|Q|$ as the average shared computation stabilizes, reaching an upper limit based on the graph's properties.

Ablation Study We evaluate key components of `ShareDP` with k=10 on the 4 largest graphs in terms of running time (Tab. 2, bold indicates the best, and underline indicates the second-best.). (1) *Merged Split-Graph Representation.* `ShareDP` outperforms `ShareDP-`, validating the effectiveness of our representation. (2) *Merged Traversal.* `ShareDP-` outperforms `maxflow` in 3 datasets, demonstrating the effectiveness of merged traversals. However, for the largest dataset, `maxflow` outperforms `ShareDP-`, highlighting the importance of well-designed components (e.g., representation of G') beyond merged traversals.

7 Conclusions

We study the problem of batch-kDP and propose the `ShareDP` algorithm. `ShareDP` shares computations across queries by consolidating converted graphs into a shared structure and sharing the traversals within this framework. Extensive experiments confirm the superiority of `ShareDP` over existing approaches.

Acknowledgement

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303. Lei Zou is the corresponding author of this work.

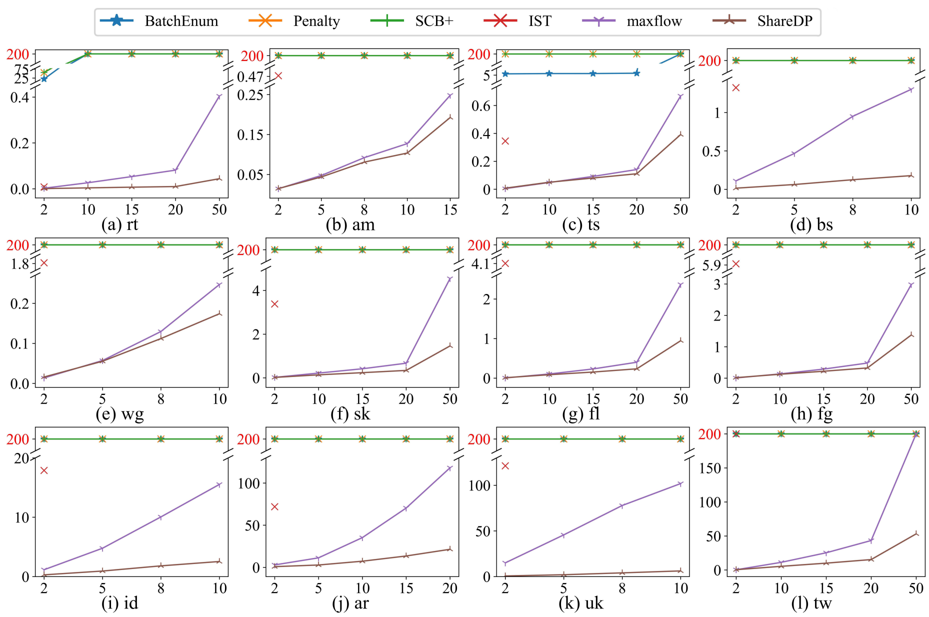


Fig. 3: Running time when varying k.

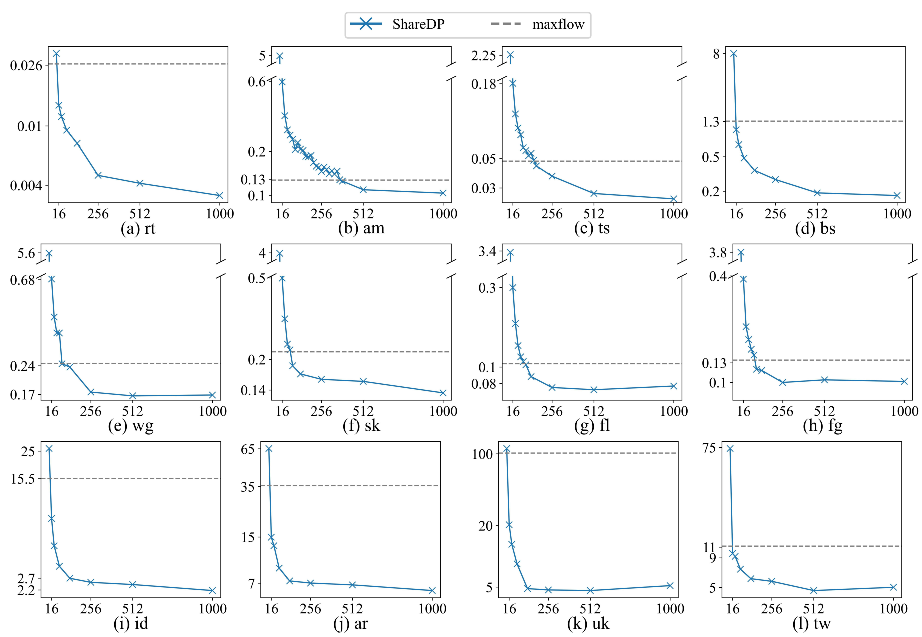


Fig. 4: Running time when varying the number of kDPs with k=10.

Table 2: Ablation Study. Average execution time (seconds) with k=10.

Method	id	ar	uk	tw
ShareDP-	6.27	19.16	26.77	19.42
ShareDP	**2.54**	**7.27**	**6.16**	**5.34**
maxflow	15.53	35.24	101.89	11.19

References

[1] B. Cheng et al. “Independent Spanning Trees in Networks: A Survey”. In: *ACM Comput. Surv.* 55.14s (2023), 335:1–335:29.

[2] T. Chondrogiannis et al. “Finding k-shortest paths with limited overlap”. In: *VLDB J.* 29.5 (2020), pp. 1023–1047.

[3] S. Curran et al. “Finding Four Independent Trees”. In: *SIAM J. Comput.* 35.5 (2006), pp. 1023–1058.

[4] S. Even et al. “Network Flow and Testing Graph Connectivity”. In: *SIAM J. Comput.* 4.4 (1975), pp. 507–518.

[5] I. T. Frisch. “An algorithm for vertex-pair connectivity”. In: *International Journal of Control* 6.6 (1967), pp. 579–593.

[6] G. Jayavelu et al. “Maintaining colored trees for disjoint multipath routing under node failures”. In: *IEEE/ACM Trans. Netw.* 17.1 (2009), pp. 346–359.

[7] L. Li et al. “Fast Query Decomposition for Batch Shortest Path Processing in Road Networks”. In: *ICDE 2020*. IEEE, 2020, pp. 1189–1200.

[8] L. Li et al. “Comparing Alternative Route Planning Techniques: A Comparative User Study on Melbourne, Dhaka and Copenhagen Road Networks”. In: *IEEE Trans. Knowl. Data Eng.* 34.11 (2022), pp. 5552–5557.

[9] A. Liu et al. “Secure and energy-efficient disjoint multipath routing for WSNs”. In: *IEEE Trans. Veh. Technol* 61.7 (2012), pp. 3255–3265.

[10] A. Mhedhbi et al. “Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins”. In: *ACM Trans. Database Syst.* 46.2 (2021), 6:1–6:45.

[11] Y. Peng et al. “Efficient Hop-constrained s-t Simple Path Enumeration”. In: *VLDB J.* 30.5 (2021), pp. 799–823.

[12] Y. Shiloach et al. “Finding two disjoint paths between two pairs of vertices in a graph”. In: *Journal of the ACM (JACM)* 25.1 (1978), pp. 1–9.

[13] J. W. Suurballe. “Disjoint paths in a network”. In: *Networks* 4.2 (1974), pp. 125–145.

[14] M. Then et al. “The More the Merrier: Efficient Multi-Source Graph Traversal”. In: *Proc. VLDB Endow.* 8.4 (2014), pp. 449–460.

[15] L. Yuan et al. “Batch Hop-Constrained s-t Simple Path Query Processing in Large Graphs”. In: *ICDE 2024*. IEEE, 2024, pp. 2557–2569.

[16] Z. Yuan et al. *ShareDP: Finding k Disjoint Paths for Multiple Vertex Pairs.* 2025. arXiv: `2502.16506 [cs.DB]`.