PeeK: A Prune-Centric Approach for K Shortest Path Computation

Wang Feng
University of North Texas

Shiyang Chen
Rutgers, The State University of New Jersey

Hang Liu
Rutgers, The State University of New Jersey

Yuede Ji
University of North Texas

ABSTRACT

The K shortest path (KSP) algorithm, which finds the top K shortest simple paths from a source to a target vertex, has a wide range of real-world applications, e.g., routing, vulnerability detection, and biology analysis. While the top K shortest simple paths offer invaluable insights, computing them is time-consuming. For example, on a Twitter graph (61.6M vertices and 1.5B edges), the best parallel method needs about 20 minutes to get 128 shortest paths between two vertices. A key observation we made is existing works search K shortest paths from the original graph, while top K shortest paths only cover a meager portion of the original graph, e.g., less than 0.001% on a Twitter graph for $K = 128$.

This paper introduces PeeK, a pruning-centric approach for KSP computation. First, PeeK applies K upper bound pruning to prune the vertices and edges that will not appear in any of the K shortest paths. Second, PeeK adaptively compacts the graph that, not only removes the deleted vertices or edges but also efficiently computes the downstream task. Furthermore, we design efficient techniques to parallelize and distribute PeeK. We compare PeeK with five algorithms on various graphs. For parallel computation with 32 threads, PeeK achieves 5.1× and 28.8× speedup over the state-of-the-art for $K = 8, 128$, respectively. More importantly, when K increases, the runtime of PeeK is barely affected. In particular, when K increases from 2 to 128 (64×), the runtime of PeeK only increases 1.1×, while the state-of-the-art method increases 10.3×.

ACM Reference Format:
Wang Feng, Shiyang Chen, Hang Liu, and Yuede Ji. 2023. PeeK: A Prune-Centric Approach for K Shortest Path Computation. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607110

1 INTRODUCTION

Finding the shortest path between two vertices in a graph is one of the most fundamental algorithms for many graph applications, e.g., navigation [55, 57, 58], biology analysis [13, 51], social network analysis [17, 24, 33, 41, 42, 52, 53], and location and traffic analysis [54, 71]. Whereas finding the shortest path is often too rigid to be practical for many real-world problems (see examples below). Therefore, the K shortest simple path (KSP) algorithm, which identifies multiple top shortest paths between two vertices, is introduced [23, 36, 70]. Generally speaking, real-world problems could have either graphs with noisy data or applications with ill-defined constraints, or both. Therefore, human experts are often required to perform analysis based on the outputs from the shortest path computation. It will be appealing if multiple top choices are computed for the experts to analyze, as that might offer more hints for them to derive the final optimal solution. Below we discuss four real-world applications of KSP:

Routing. The network routing problem aims to select the best path for navigation. The KSP algorithm is often used to find multiple paths [45, 66]. For instance, in an optical transport network, a KSP-based routing and spectrum assignment algorithm can be used to enable a flexible optical path network [66]. In particular, it first finds K ordered paths with the KSP algorithm. Then, it iteratively checks the availability of the paths in increasing order. The first available one will be the assigned path. Recently, the KSP algorithm is also applied to the low earth orbit satellite networks (LSNs) [29], e.g., SpaceX Starlink [8], and Amazon Kuiper [26].

Vulnerability detection. In code vulnerability detection, one usually needs to find K shortest paths on a control flow graph (CFG) to verify the existence of a vulnerability [37, 39, 40, 69, 72]. In particular, a code site that an attacker can control is denoted as a source vertex while another code site with vulnerability is denoted as a target vertex in the CFG. To determine whether an attacker can exploit this vulnerability, one needs to verify whether the conditions on the paths between the source and target can be satisfied. Since one cannot afford to find all the paths between them, existing works [40, 69, 72] often identify top K most likely exploitable (shortest) paths and verify. Note that only finding the single-source shortest path is also not desirable as that will likely miss many vulnerabilities.

Biology analysis. In biology, various types of interaction data are represented by a graph, e.g., protein-protein interaction, protein-DNA/RNA interaction, and genetic interaction [50, 62]. The KSP algorithm is often used to analyze complex interactions as multiple shortest paths are required. For example, in a gene interaction network, a vertex represents a gene and an edge denotes their interaction. The gene inference problem is to identify the potential regulatory pathways passing through a gene, where a regulatory pathway is denoted as a path of interacting genes from a causal gene to a target gene. The KSP algorithm can identify multiple such paths which are the potential pathways [50, 62].

Graph database is an increasingly popular type of NoSQL system, often using the property graph data model [9]. KSP search is

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607110

one of the key features of the newly proposed GQL query languages for property graph systems, as well as the SQL/PGQ extension, which is new in SQL:2023 and proposed and backed by the International Organization for Standardization (ISO) [20]. Particularly, two types of KSP are included in both GQL and SQL/PGQ. One is the exact KSP, named `SHORTEST k`. The other is `SHORTEST k GROUP`, a KSP variant by grouping the paths with the same lengths and returning the k shortest groups.

1.1 Existing Solutions

A naive solution is finding all the simple paths between the two vertices and then extracting the K shortest. However, this is impractical as finding all the simple paths is an NP-hard problem [61]. Therefore, various dedicated KSP algorithms have been proposed.

Yen's algorithm is the foundational algorithm of the KSP problem [70]. In particular, it first finds the shortest path from the source vertex s to the target vertex t via a single-source shortest path (SSSP) algorithm. Next, it iterates K-1 times, and each iteration will find the next shortest path. In particular, in the i-th iteration, it takes the previously derived (i-1)-th shortest path as the deviation path. Later, it takes each vertex on that path as a deviation vertex (denoted as v), and the subpath from source S to v as the prefix path. The goal is to find a different shortest path from v to the target vertex T and concatenate that path with the prefix path. This concatenated path is subsequently added to the candidate path set. After iterating all the vertices in the (i-1)-th shortest path, the shortest one from the candidate set is the i-th shortest path. The time complexity of Yen's algorithm is $O(Kn(m+n\log n))$, where n and m denote the number of vertices and edges, respectively.

Yen's algorithm is computationally intensive since it runs a large number of the SSSP algorithms, i.e., up to Kn. To improve its performance, two major types of algorithms have been proposed. (i) **Node classification (NC) algorithm** [25] maintains a reverse shortest path (SP) tree from every vertex to the target. It also classifies the vertices into three colors, i.e., red, yellow, and green. A vertex is red if it is on the prefix path, green if it can reach the target vertex without visiting a red vertex, and yellow as the remaining vertices. For each deviation vertex, it can directly identify a candidate shortest path using the reverse SP tree if its neighbor vertex with the lowest weight is green. Otherwise, it runs SSSP on a subgraph only with the yellow vertices. Though it reduces the number of SSSP calls, it introduces additional overheads in updating the reverse SP tree and vertex colors. OptYen [5] avoids the dynamic updates by only using one reverse SP tree and computing the SSSP on the original graph if needed.

(ii) **Sidetracks-based (SB) algorithm** [47] records a large number of reverse SP trees from different vertices and aims to reduce the SSSP calls by reusing those trees to find a valid shortest path. This algorithm has an obvious memory issue as it stores a large number of reverse SP trees. SB* algorithm further improves the time efficiency by resuming the previously computed SSSP when a new SSSP needs to be computed [6, 7]. SB* is considered the state-of-the-art serial algorithm though it costs even more space to record the status of the previously computed SSSPs.

For **parallel computation**, existing works have explored two strategies. One is directly parallelizing the SSSP algorithm [5, 63]. The other aims to parallelize the execution of multiple SSSPs [63]. In particular, on each deviation path, there is no dependency between all the SSSPs starting from different source vertices. Combining the two strategies, Ajwani *et al.* proposed a hybrid method [5], which is able to not only concurrently run multiple SSSPs but also run each SSSP in parallel. With the hybrid parallel strategy, **OptYen** is considered the state-of-the-art parallel method [5].

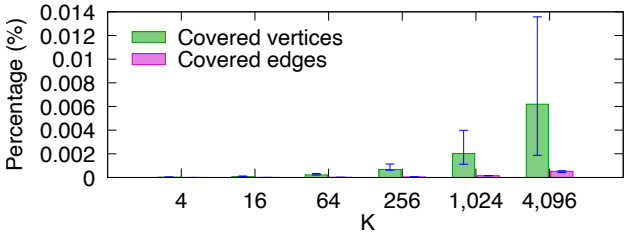


Figure 1: The percentage of the covered vertices and edges against different K values for a Twitter (GT) graph with 61.6M vertices and 1.5B edges.

1.2 Observation

Due to the high time complexity, it requires an enormous amount of time to compute KSP, even for a small K value. For example, for the popular Twitter graph (61.6M vertices and 1.5B edges) [11], the best parallel KSP method, OptYen [5], would require, on average, 1,131 seconds (or about 20 minutes) to compute 128 shortest paths between two vertices. This time is projected to grow significantly for larger graphs and K. It is hence of paramount value to accelerate the KSP algorithm such that the top K shortest paths can be derived in an acceptable time envelope.

A key observation we found is **the vertices from the top K shortest paths only cover an extremely small portion of the original graph, even for very large K values.** However, existing works always derive the K shortest paths on the original graph, potentially leading to redundant computations. Figure 1 shows the percentage of the covered vertices and edges for different K values on the Twitter graph on an average of 100 randomly selected source and reachable target vertices. Here, a vertex or edge is covered if it appears in any of the K shortest paths. Even for a K value as large as 4,096, the covered vertex percentage is less than 0.01%, and the edge is less than 0.001%. Among all the source and target pairs, the maximum percentage of the covered vertices is less than 0.014%, and the edge is less than 0.001%. That means only 3,813 out of 61.6 M vertices appear in the final 4,096 shortest paths. The same observation also holds for other tested graphs (Section 7).

1.3 Contributions

In this paper, we devise PeeK[1][2], a new method that can remarkably accelerate the computation for KSP problems. Given a graph, a source vertex s, a target vertex t, and K, PeeK first applies the K upper bound pruning technique to prune the vertices and edges that will not appear in any of the K shortest paths. Later, PeeK judiciously chooses two compaction methods to remove the deleted

[1] PeeK stands for prune-centric KSP.

[2] The source code of PeeK is available at https://github.com/SC-Lab-Go/PeeK

vertices and edges for better memory access during KSP computation and lower compaction overhead. Finally, PeeK computes the KSPs with an optimized node classification method.

K upper bound pruning. Starting from the key observation, we propose a K upper bound pruning technique. *The key idea is to remove the unnecessary vertices and edges as many as possible, where a vertex or edge is unnecessary if it will not appear in the K shortest paths.* To achieve that, this method prunes the vertices and edges whose shortest distance from the source to the target passing this vertex is longer than an estimated upper bound of K shortest path distance. Deriving a tight and sound upper bound is the key to this approach, which is done as follows: we run two SSSPs to derive the shortest distances from the source vertex to each vertex and from each vertex to the target vertex. Subsequently, we add these two distances to derive the shortest paths from the source through each vertex to the target vertex. Finally, for the vertices that fail to appear in the estimated valid K shortest paths, we delete them as well as their edges. This pruning can help reduce 98.4% vertices and 97.7% edges on average across eight tested graphs when K equals 8. Similar pruning powers are observed for other K values.

Adaptive graph compaction. The K upper bound pruning will update the graph several times by deleting vertices or edges. To efficiently update the graph and compute the downstream task, we design an adaptive graph compaction method that takes the efficient static graph format, i.e., compressed sparse row (CSR) [15], as the base and adaptively selects from two strategies. One is edge swap-based, which swaps the deleted edges of a vertex to the end of its CSR representation. The other is graph regeneration based, which regenerates a new CSR for the remaining graph. Compared with the K upper bound pruning using the conventional status array method to compact the graph, the adaptive strategy can bring 1.5× and 33× speedup for K equaling 8 and 128, respectively.

Parallel computation. To efficiently compute PeeK in both shared and distributed memory systems, we first classify all the jobs as data, embarrassingly, or task parallel. As embarrassingly and task parallel jobs incur low communication overhead, we aim to leverage them the most for parallel design. That is, K upper bound pruning is mostly designed as a data parallel job, the adaptive graph compaction is an embarrassingly parallel job, and the KSP computation is a task parallel job. With a few specialized optimizations for both shared- and distributed-memory computation, PeeK can achieve stable scalability. In particular, for shared-memory systems, PeeK with 32 threads achieves 4× speedup over 1 thread. For distributed-memory systems, PeeK with 64 computing nodes (1,024 cores) achieves 30× speedup over 1 computing node (16 cores).

Experiment. We compare PeeK with five algorithms, including Yen's algorithm [70], node classification (NC) [25], OptYen [5], sidetracks-based (SB) [7], and an optimized SB algorithm, named SB* [7]. Among them, OptYen is the state-of-the-art parallel method, and SB* is the state-of-the-art serial algorithm. On eight tested graphs, for serial computation, PeeK achieves 2.2× and 3.1× speedup over the state-of-the-art for $K = 8, 128$, respectively. For parallel computation with 32 threads, PeeK achieves 5.1× and 28.8× speedup over the state-of-the-art for $K = 8, 128$, respectively. More importantly, when the value of K increases, PeeK is barely impacted. That is, when K increases from 2 to 128 (64×), PeeK only increases 1.1×, while the state-of-the-art method increases 10.3×.

Algorithm 1: Yen's algorithm.

```
Function YenKSP(G, s, t, K):
1  P[0] = SSSP(G, s, t) ;                    // The found shortest path.
2  C = ∅ ;                                   // The candidate path set.
3  for k ∈ [1, K) do
4      for vertex v ∈ P[k − 1] do
5          prefix = subpath(s, v, P[k − 1])
6          remove any edge v-w (w ∈ N(v)) if it ∈ P[0], ..., P[k − 1]
7          suffix = SSSP(G, v, t)
8          newPath = prefix + suffix
9          C.add(newPath) if newPath ∉ C
10         restore graph G
11     P[k] = min(C) if C ≠ ∅
12 return P
```

The **Novelty of PeeK** lies in three aspects. (i) *K upper bound pruning* is the key contribution, which is first introduced to KSP computation. (ii) *Adaptive graph compaction* is used to couple K upper bound pruning as the remaining graph might change dramatically, which should be computed differently to gain the best performance. (iii) *PeeK can integrate with existing KSP algorithms* to boost their performance. In particular, K upper bound pruning can serve as a preprocessing step for existing algorithms. Also, one can fuse the adaptive compaction method to achieve the best performance.

2 BACKGROUND

Problem definition. Formally, we define KSP as below.

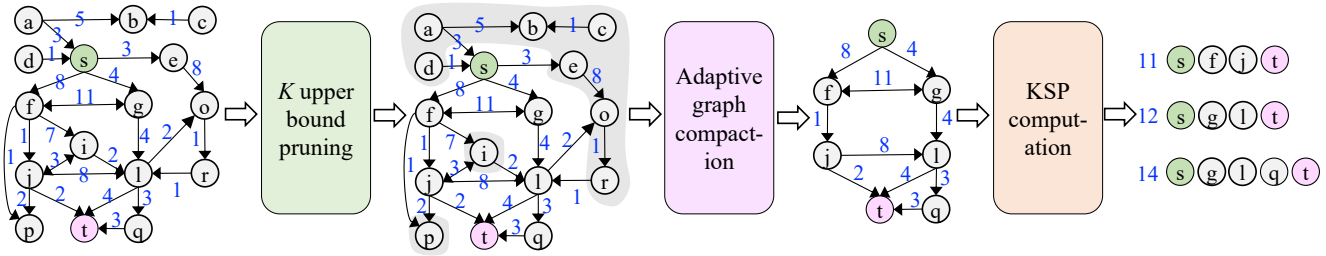
Definition 1. *Let $G(V, E, W)$ be a directed and weighted graph, where V, E, and W denote the vertex, edge, and edge weight sets, respectively. We require $\{w > 0 | \forall w \in W\}$. Given a source vertex S, a target vertex T, and the required number of shortest paths K, the KSP algorithm finds K paths that meet two conditions, that is, these paths are (i) loopless and (ii) the top K shortest distances. Here, the path distance is the sum of the edge weights on that path.*

Yen's algorithm [70], shown in Algorithm 1, runs multiple SSSPs (lines 1 and 7) and dynamically updates the graph (lines 6 and 10) to make sure the previously found shortest paths are excluded. After iterating K-1 iterations (line 3), the final K shortest paths will be identified.

3 OVERVIEW

Figure 2 overviews PeeK with a running example. Particularly, first, PeeK performs K upper bound pruning to quickly identify the unnecessary vertices and edges that will not appear in any K shortest paths. This is motivated by the observation that K shortest paths use only a small portion of the vertices and edges. Here, vertices $\{a, b, c, d, e, i, o, p, r\}$ are identified to be pruned. Later, PeeK adaptively compacts the graph by pruning the unnecessary vertices and their adjacent edges. PeeK judiciously selects between edge swap and graph regeneration-based graph compaction techniques. After that, the compacted remaining graph is shown in Figure 2(c).

In the end, we compute the KSPs on the compacted graph by customizing the state-of-the-art parallel method, i.e., OptYen [5].



(a) An example graph (b) Identified unnecessary vertices and edges (c) Compacted graph (d) KSP results

Figure 2: Overview of PeeK. Given an example graph shown in (a), source vertex s, target vertex t, and K equals 3, PeeK first applies K upper bound pruning to identify the unnecessary vertices and edges as shaded in (b). Next, PeeK adaptively compacts the graph to be (c) by pruning the unnecessary vertices and edges. Further, PeeK computes KSPs as shown in (d).

In particular, we only use the reverse tree and disregard the vertex colors. The reverse tree can help quickly find a candidate path. If such a path is simple, no further computation is required. Otherwise, we compute a new SSSP on the remaining graph. In this example, the identified three shortest paths are shown in Figure 2(d).

4 GRAPH PRUNING WITH K UPPER BOUND

4.1 K Upper Bound Pruning

The key insight of K upper bound pruning is, *a vertex v can be pruned if the shortest distance from the source to the target passing this vertex, denoted as $dist(s, v, t)$, is longer than the estimated K shortest paths.* This is true as the other paths passing v will not be shorter than $dist(s, v, t)$. Similarly, the edges adjacent to vertex v will not appear on any K shortest paths and therefore can also be pruned.

Algorithm 2 denotes the pseudocode of the proposed K upper bound pruning technique. The key challenge is to *efficiently* find a *correct* and *tight* upper bound, denoted as b. It must be correct since we cannot overly prune any vertex or edge that appears in the final K shortest paths. Also, it should be tight so we can prune as many unnecessary vertices or edges as possible. The ideal upper bound is the distance of the K-th shortest path. Though it can not be obtained at this stage, we can leverage the shortest distance of any vertex v, i.e., $dist(s, v, t)$, to get a correct and tight upper bound. That is, we can first get the shortest distance from the source vertex and to the target vertex for all the vertices, i.e., $spSrc[*]$ and $spTgt[*]$ (lines 1-2 in Algorithm 2). Then, we can obtain the shortest distance by adding up the two distances. In particular, for any vertex v, the shortest distance from the source to the target vertex passing it is greater than or equal to $spSrc[v] + srTgt[v]$ (lines 3-4 in Algorithm 2). This is summarized in Lemma 4.1 and proved accordingly.

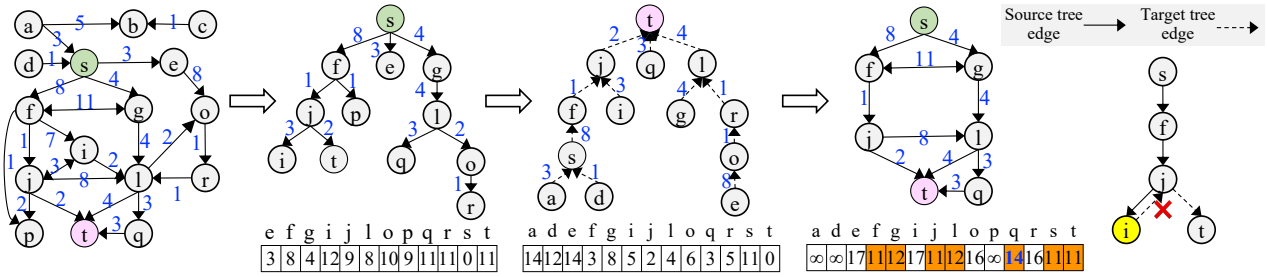
Algorithm 2: K upper bound pruning algorithm.

```
Function kUpperBoundPruning(G, s, t, K):
  // Step 1: Get shortest distances from source and to target.
1 spSrc, parentSrc = SSSP(G, s)
2 spTgt, parentTgt = reverseSSSP(G, t)
  // Step 2: Identify K upper bound.
3 for v ∈ G in parallel do
4     dist[v] = spSrv[v] + spTgt[v]
5 q = ∅
6 for v ∈ G in the increasing order of dist[*] do
7     q.push(v) if isValid(v, parentSrc, parentTgt)
8     break if q.size() == K
9 b = dist[q.back()]
  // Step 3: Prune unnecessary vertices and edges.
10 for v ∈ G in parallel do
11     G.delete(v) if dist[v] > b
12 for e ∈ G in parallel do
13     G.delete(e) if G.weight(e) > b
14 return G
```

Lemma 4.1. *For any vertex v, the distance from the source s to the target t via vertex v is equal to or longer than $spSrc[v] + srTgt[v]$.*

Proof. We use $s \rightarrow ... \rightarrow v \rightarrow ... \rightarrow t$ to denote the combined path from $spSrc[v] + srTgt[v]$. Since $spSrc[v]$ denotes the shortest distance from s to v, the path $s \rightarrow ... \rightarrow v$ is the shortest. Similarly, the path $v \rightarrow ... \rightarrow t$ is also the shortest. Therefore, if no duplicate vertex appears, this path will be the shortest from s to t passing vertex v. However, if there are duplicate vertices, we need to find an alternative path either from s to v or from v to t to find the correct shortest path, which will be greater than or equal to the distance in $spSrc[v]$ or $spTgt[v]$. Together, we proved it. □

Figure 3(b) and (c) show the generated source and target SSSP trees along with *spSrc* and *spTgt*, respectively, for the running example graph. However, the path generated in this way may not be valid as it might include loops, because the subpath from the source SSSP tree may intersect with the subpath from the target SSSP tree. For example, as shown in Figure 3(e), the source path of vertex i is $s \rightarrow f \rightarrow j \rightarrow i$, while the target path is $i \rightarrow j \rightarrow t$. The combined path is invalid as vertex j appears twice. To address that, we design a path validation check function to see whether it is valid. In particular, we first find the subpath from the source vertex by backtracking the parent array. Later, we apply the same strategy to find the subpath to the target vertex. From these two subpaths, a path is regarded as valid if there is no duplicate vertex.



(a) An example graph. (b) The ***source*** SSSP tree and distance array ***spSrc***. (c) The ***target*** SSSP tree and distance array ***spTgt***. (d) The remaining graph and distance sum array ***spSum***. (e) The invalid path for node ***i***.

Figure 3: The process of K upper bound pruning when $K = 3$. (b) denotes the source SSSP tree and the distance array $spSrc$. (c) denotes the target SSSP tree and the distance array $spTgt$. (d) shows the distance sum array $spSum$, the identified K upper bound value 14, and the remaining graph. (e) denotes an invalid path taking vertex i as an intermediate vertex.

After we get the shortest, valid, and unique K paths, we can use that distance as the K upper bound value, b. To get that, we explore the vertices in the increasing order of their distances, check if they are valid, and make sure the path is unique, which means no two same paths can be added to the queue (lines 6-8). We will take the K-th valid distance as the upper bound value, b (line 9). Using Figure 3 as an example, after investigating the shortest path starting from vertex s, we can skip vertex f, j, and t as they share the same path. By only investigating three paths passing s, g, and q, we derive the upper bound value 14. Subsequently, we can prune the vertices whose distances are greater than the upper bound (lines 10-11). This is summarized in Lemma 4.2 and proved accordingly. In the last, we further prune the edges whose weights are greater than the upper bound value (lines 12-13).

Lemma 4.2. *Given the estimated upper bound value b, if the shortest path from the source vertex s to the target vertex t via vertex v is greater than b, vertex v cannot appear in any K shortest paths.*

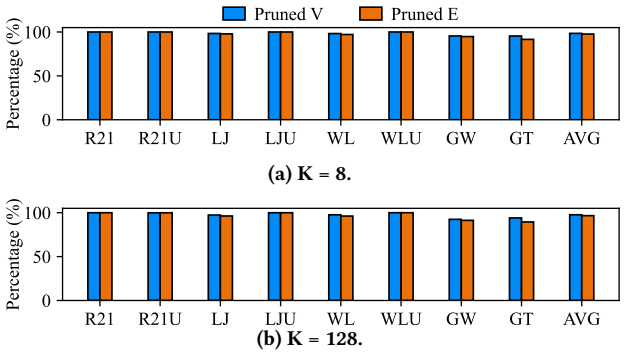
Proof. This Lemma holds true since there are already K shortest paths covered by the upper bound value b. □

As shown in Figure 3, from the two shortest distance arrays, $spSrc$ and $spTgt$, we can get the shortest distance passing each vertex, denoted as $spSum$. From there, we derive the upper bound value b as 14 when K equals 3. Then, we can prune all the vertices whose $spSum$ is greater than the upper bound. With that, the vertices {a, b, c, d, e, i, o, p, r} along with their edges are pruned. Note that, the vertices {b, c} are pruned as they are unreachable.

Theorem 4.3. *The K shortest paths found from the pruned graph are the same as the ones from the original graph.*

Proof. Compared with the original graph, we pruned the unnecessary vertices and edges based on the K upper bound value. According to Lemma 4.2, only the vertices that are not in any K shortest paths are pruned, which means, the vertices covered by the K shortest paths are kept. Therefore, the K shortest paths found from the pruned graph are the same as the ones found from the original graph. □

This Theorem presents the soundness of our design.



(a) K = 8.

(b) K = 128.

Figure 4: Percentage of pruned vertex (V) and edge (E) by K upper bound pruning for eight tested graphs and the average when K is (a) 8 and (b) 128.

4.2 Benefits and Complexity Analysis

Benefits. Figure 4 presents the percentage of pruned vertex and edge count for eight tested graphs (Section 7). For each graph, we run 32 pairs of randomly chosen source and reachable target vertices. One can observe that our K upper bound pruning technique can prune a large percentage of the vertices and edges, which can dramatically reduce the required graph for KSP computation. In particular, when k equals 8, it prunes 98.4% vertices and 97.7% edges on average. When K becomes as large as 128, it shows a similar pruning power by pruning 97.7% vertices and 96.6% edges.

Complexity analysis. The time complexity of K upper bound pruning is $O(m+(K+\lambda+\log n)n)$, where n and m denote the vertex and edge count, respectively, K denotes the required number of shortest paths, and λ denotes the number of inspected invalid paths. This time complexity is made up of three major operations.

First, the classical single-source shortest path (SSSP) algorithm takes $O(m+n\log n)$ time [28]. Second, the vertex and edge deletion operation takes $O(m+n)$ time as it needs to iterate all the vertices and edges. Third, finding the K upper bound value takes $O((K+\lambda+\log n)n)$ time. It needs to sort the vertices with $O(n\log n)$. Further, it needs to verify the validity for $K+\lambda$ paths and each validity verification takes $O(h)$, where h denotes the height of the shortest path tree. Combining them together, we get the time complexity

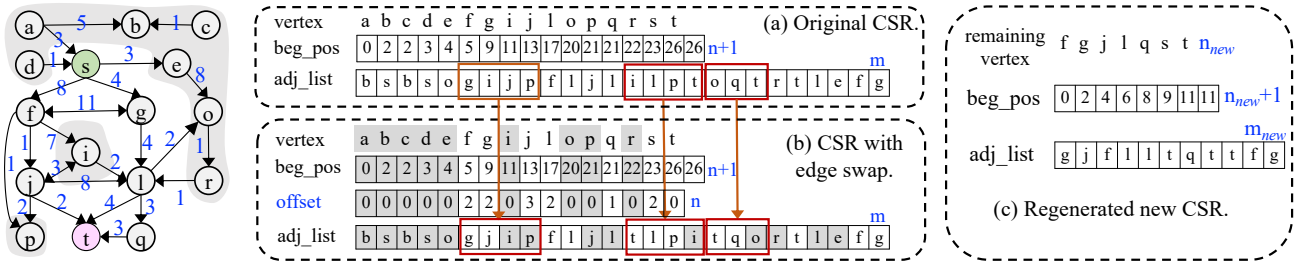


Figure 5: For the running example graph, (a) shows the original CSR, (b) shows the CSR after edge swap-based graph compaction where the shaded vertices and edges are deleted, and (c) shows the new CSR after graph regeneration. The highlighted edges in the adjacency list from (a) to (b) show the changes made by the edge swap-based method.

of $O(m + (K + \lambda + \log n)n)$. In practice, we observe the value of λ is small, and usually less than K. Compared to the KSP algorithm with $O(Kn(m + n \log n))$ time complexity, K upper bound pruning is much faster by taking about $O(1/Kn)$ time of the KSP algorithm.

The space complexity of K upper bound pruning is $O(n)$ as it uses five arrays with the size of n. That includes the shortest distance and parent arrays both from the source and to the target, and another distance array representing the sum.

5 ADAPTIVE GRAPH COMPACTION

5.1 Unique Graph Compaction Patterns

The graph is updated three times during the implementation, i.e., after the first SSSP, after the second SSSP, and after the K upper bound pruning. After the first SSSP, we can prune the vertices and edges that are unreachable from the source. This can help to improve the computational efficiency of the second SSSP. Similarly, after the second SSSP, we can prune the vertices and edges that can not reach the target. We observe two unique computation patterns of the graph update: (i) It only involves vertex or edge deletion, not insertion. That also means the deleted vertex or edge will not be recovered. (ii) The graph is updated only three times in batch style.

More importantly, various heavy computations will be performed after the graph update, including SSSPs, sorting, and KSP. That means we need to focus on the end-to-end computation time, which includes both graph update and downstream computation. Though recent works on dynamic graphs have achieved significant improvement [21, 59, 67, 68], they fall behind for the end-to-end computation in this scenario. In particular, for the Twitter graph, one of the state-of-the-art dynamic graph update systems, Terrace [59], takes about 1,900 seconds to compact the graph, while regenerating a new compressed sparse row (CSR) [15] only takes about 1 second.

The CSR format [15] is commonly used as it is not only space efficient but also locality preserved. Figure 5(b) presents the CSR format of the running example graph. In particular, it uses two arrays, one is the adjacency list of length m by saving the target vertex of each edge (*adj_list* in Figure 5(b)). The other is the beginning position array of length $n + 1$ (*beg_pos* in Figure 5(b)), where each value denotes the corresponding vertex's beginning position in the adjacency list. To traverse a vertex's (e.g., v) edges, one can easily get them from the *adj_list* with the range [*beg_pos[v], beg_pos[v+1]*).

However, the CSR format is less efficient when the graph is dynamically changing. Motivated by this, we design an adaptive compaction method that takes the CSR format as the base and adaptively updates it. In particular, we adaptively select from two strategies, i.e., edge swap and graph regeneration-based.

5.2 Edge Swap-based Compaction

The edge swap-based compaction method is specially designed for vertex/edge deletion based on the CSR format. In particular, it introduces two additional arrays, one is a vertex status array to mark whether a vertex is deleted or not, and the other is an offset array to indicate the new offset of each vertex in the adjacency list. Both of them are with lengths of n.

Edge swap-based method deletes vertex and edge in the following ways. (i) For vertex deletion, it changes the status of the deleted vertices to 0 in the status array. One can avoid visiting the deleted vertices by checking the status array first. As shown in Figure 5, the vertices $\{a, b, c, d, e, i, o, p, r\}$ are marked as deleted. (ii) For edge deletion of a source vertex, it swaps the deleted edge in the adjacency list with the last kept edge and updates the number of valid edges (*offset* array) by decreasing 1. When traversing the edges of vertex v, instead of using the *adj_list* with the range $[beg\_pos[v], beg\_pos[v+1])$ in traditional CSR, it uses the range $[beg\_pos[v], beg\_pos[v] + offset[v])$. For vertex f in Figure 5, the edges to i and p are deleted. It swaps the deleted edge to i with the kept edge to j, and the edge p does not change since it is in the end. Further, the offset value is changed to 2.

The edge swap operation for a vertex can be implemented by iterating two pointers, i.e., a front pointer from the first edge, and a back pointer from the last edge. As the edge swap-based graph compaction needs to iterate all the edges of the remaining vertices and the swap operation takes $O(1)$ time, the total time complexity is $O(n + m_a)$, where $m_a = \sum_{v \in V_r} d(v)$, V_r denotes the remaining vertex set, and $d(v)$ is the out degree of vertex v. The m_a is less than m, but greater than m_r which is the remaining edge count.

5.3 Graph Regeneration-based Compaction

Graph regeneration-based method regenerates a new CSR only with the remaining vertices and edges. To create a new CSR, we first build a vertex map between the original and new vertex ID. For the example in Figure 5(c), the new ID of vertex f is 0, which was 5 in the original CSR. Then, we iterate the remaining vertices and their edges to calculate the new offset in the new adjacency list. For example, the offset of vertex f becomes 2 as only the edges to g and j are remaining. In the end, we update the new beginning position array and the new adjacency list with the new vertex IDs.

Figure 5(c) shows the newly generated CSR for the example graph. In particular, the lengths of the beginning position array and adjacency list become 8 and 11, respectively, which were 17 and 26 before. With that, the downstream computation will be solely based on the new CSR representation. The time complexity of the graph regeneration-based method is the same as the edge swap-based method, i.e., $O(n+m_a)$, since we need to iterate all the vertices and all the edges of the remaining vertices.

5.4 Adaptive Compaction

Our adaptive selection method judiciously switches between edge swap-based and regeneration-based graph compaction based on the end-to-end performance, including both graph compaction and the downstream computation task.

Comparison of graph compaction. The edge swap-based graph compaction method will update the status array for the deleted vertices, which equals $n - n_r$, where n_r denotes the number of remaining vertices. Then, it traverses the remaining vertices in the original CSR, and traverses the edges of all the remaining vertices (m_a). In the worst case, it needs to traverse all the edges. That means, *its runtime is positively correlated to the edge count of all the remaining vertices* m_a. In contrast, the regeneration-based method takes a longer time to compact the graph as it involves several operations. Creating a vertex map will read n times, while the update takes n_r times. Calculating the new offset in the adjacency list will read n vertices and m_a edges, and the update to the new CSR takes $m_r + n_r$ times. In total, the reading takes $m_a + 2n$ times, and the update takes $m_r + 2n_r$ times. That means, *its runtime is positively correlated to the number of remaining vertices and edges.*

Observation I: *The graph compaction of both methods is positively correlated to the number of remaining vertices and edges, while the regeneration-based method is slower.*

Comparison of downstream computation. For the downstream computation task, the edge swap-based method still uses the original CSR though with a status array and a new *offset* array. Though the time complexity of the downstream computation is based on the remaining graph, the actual runtime is decided by the number of remaining vertices and edges. That means, *its runtime is positively correlated to the number of remaining vertices and edges.* For the regeneration-based method, the downstream computation is based on the new CSR. Assuming KSP is the downstream task, the time complexity will become $O(Kn_r(m_r+n_r \log n_r))$, where m_r and n_r denote the edge and vertex count for the remaining graph. It also enjoys better data locality than the edge swap-based method as no deleted vertices or edges are in the CSR. That means, *the more vertices or edges remaining, the greater* m_r *and* n_r *become, thus more computation will be required.*

Observation II: *The downstream computation of both methods is positively correlated to the number of remaining vertices and edges, while the regeneration-based method is faster.*

Adaptive selection. Combining Observation I and II, we can conclude that when the remaining graph is large, e.g., in the same order as the original graph, the edge swap-based method fits better as it takes much less time to compact the graph. Conversely, when the remaining graph is smaller, the graph regeneration-based method fits better as its downstream computation is much faster. Motivated by this, we quantitatively estimate the difference, i.e., $m_r < \alpha \cdot m$, where m_r and m denote the edge count of the remaining and original graph, respectively, and α is a coefficient to estimate the trade-off between the two methods. That is, when $m_r < \alpha \cdot m$, we select the regeneration-based method, otherwise edge swap-based method. α is in $[0, 1]$, which is mainly decided by the downstream task, i.e., a heavier workload suggests a higher value (e.g., 0.6) so that the regeneration-based method can be more likely used.

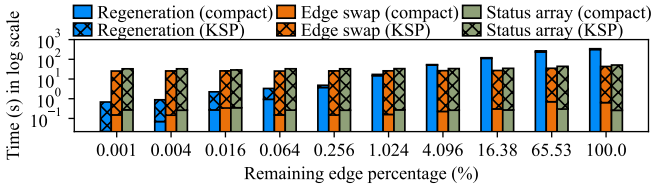


Figure 6: The end-to-end performance of graph regeneration, edge swap, and status array-based methods plus the downstream KSP ($K = 8$) computation on the Twitter graph.

Figure 6 shows an example with the Twitter (GT) graph for the end-to-end time difference between the two methods when the number of remaining edges varies. We also compare with a baseline method, i.e., status array-based, which uses two status arrays to mark whether a vertex or edge is deleted or kept. We first make sure the vertices and edges in the K shortest paths ($K = 8$) are always kept, then we randomly delete the required number of edges. We start by only keeping 0.001% percentage of edges, and increase by 4 times till the whole graph is kept.

One can find four interesting observations. (i) When the kept graph is extremely small, the regeneration-based method is much faster than others, e.g., 48× and 37× speedup over the status array and edge swap-based, respectively, for 0.001% edges. Then, the runtime of the regeneration-based method gradually grows with the increase of the remaining edges. (ii) When the edge count is between 1% and 16%, the three methods share a similar time. (iii) When the majority of the graph is kept, edge swap-based is much faster than regeneration-based, e.g., 4.4× and 7.6× speedup for 65.53% and 100%, respectively. (iv) Further, the edge swap-based method is faster than the status array-based method. In particular, it consistently achieves about 1.3× speedup for different edge percentages.

6 PARALLEL PEEK

6.1 Parallel Computation Patterns of PeeK

Figure 7 summarizes the parallel computation patterns of PeeK. We classify each job as data parallel, embarrassingly parallel, or task parallel. *Data parallel job* runs the same task on different parts of the data simultaneously with all the workers. *Embarrassingly parallel job* is a subset of task parallel workloads, where there is little or no dependency for communication between different workers. *Task parallel job* refers to one or more independent tasks that can be running concurrently. As embarrassingly and task parallel jobs usually incur low communication overhead, we aim to leverage them the most for the parallel and distributed design of PeeK.

K **upper bound pruning** includes two major components, i.e., SSSP computation, and identifying K upper bound value. (i) The

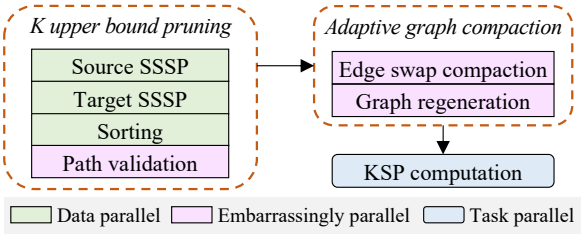


Figure 7: The parallel computation patterns of PeeK.

SSSP can be computed with parallel SSSP algorithms, which is a *data parallel job*. It is computed twice, one is from the source vertex with outgoing edges, and the other is from the target vertex with incoming edges. (ii) In identifying the K upper bound value, sorting the distance sum array can be implemented with parallel sorting algorithms [1, 35, 64]. Further, for path validation, computing if any vertex from the source path also appears in the target path takes most of the time. After getting the two paths, we design an *embarrassingly parallel* method by concurrently searching for any vertex in the source path against all the vertices in the target path. We use a hash table to achieve $O(1)$ search time complexity.

Adaptive graph compaction judiciously selects between edge swap and graph regeneration-based compaction techniques. We layout both as embarrassingly parallel workloads. (i) We parallelize the edge swap-based graph compaction technique by allowing each thread to work on its own partition of the vertices. In particular, for each vertex, it will use two pointers to scan its edges in the array of *adj_list* and swap the kept vertices to the front. (ii) We parallelize the graph regeneration with embarrassing parallelism in three steps. First, we scan all the vertices by allowing each worker to compute its own partition to identify the kept ones. Second, for the kept vertices, we partition them based on the number of available threads and use the prefix sum [12, 31] to figure out the number of kept edges for each partition. Then, we write the kept edges to the new CSR by allowing each worker to compute its own partition.

For **KSP computation**, we leverage a two-level parallel strategy proposed by Ajwani *et al.* In the inner level, we can directly parallelize the frequently called SSSP algorithm [5, 63]. As the KSP computation needs to run up to Kn SSSPs, and there is no dependency between the SSSPs starting from different source vertices on the same deviation path, we concurrently run multiple SSSPs in the outer level. The outer level workload can be implemented as task parallel, while the inner level workload is data parallel.

6.2 Parallel Implementation

Shared-memory implementation. Following the parallel computation patterns of PeeK, we use the efficient Δ-stepping [56] algorithm for SSSP computation. The key idea of Δ-stepping is to group the vertices into buckets and process a bucket of vertices in parallel instead of sequentially processing one-vertex-at-a-time in Dijkstra's algorithm [22]. In identifying the K upper bound value, we use the block indirect sort algorithm [1]. For both edge swap and graph regeneration-based compaction, we use *vertex-centric parallelism* as the dependency only appears within the edges of one vertex, not crossing vertices. To avoid workload imbalance between different threads, we partition them based on the edge count to make sure each partition gets an approximately equivalent number of edges. For KSP computation on the remaining graph, assume there are p threads, and the length of the i-th deviation path is l_i, we assign $\lfloor p/l_i \rfloor$ threads for each SSSP with the ones closing to the source vertex to get one extra thread if any threads are remaining.

Distributed-memory implementation. We distribute PeeK following the same strategy of the shared-memory design. We partition the graph using row-wise 1-d partitioning [3, 16, 48]. Though it is simple, it is communication friendly and does not yield extra time for pre-processing. We leverage the distributed Δ-stepping algorithm [3] for both SSSPs. In identifying the K upper bound value, we use a distributed sample sort algorithm [2]. For adaptive graph compaction, we implement the distributed version of edge swap-based and graph regeneration-based compaction techniques as both are embarrassingly parallel tasks. For KSP computation on the compacted graph, we map the two-level parallel strategy to the distributed architecture. That is, the outer level job, i.e., one or multiple SSSPs from each deviation vertex, will be mapped to one computing node. The inner level job, i.e., the Δ-stepping algorithm, will be mapped to multiple cores inside a computing node.

Table 1: Graph benchmarks (sorted by vertex count).

Graph	Abbr.	Graph type	# vertex	# edge	Weight
Rmat21	R21	Synthetic graph	2.1M	33.6M	random
Rmat21-U	R21U	Synthetic graph	2.1M	33.6M	1
Livejournal	LJ	Social network	4.8M	68.5M	random
Livejournal-U	LJU	Social network	4.8M	68.5M	1
Wikipedia	WL	Article network	13.6M	437.2M	random
Wikipedia-U	WLU	Article network	13.6M	437.2M	1
GAP-web	GW	Web network	50.6M	1.9B	real
GAP-twitter	GT	Social network	61.6M	1.5B	real

7 EXPERIMENT

7.1 Experimental Setup

We implement PeeK with over 5,000 lines of C++ code. PeeK is compiled by GCC with O3 optimization level. We use OpenMP as the multithreading library and OpenMPI as the message-passing interface library. The single-machine experiments are performed on a server with two Intel Xeon Silver 4309Y CPUs running Rocky Linux 8.6. For distributed experiments, we run them on the servers of Texas Advanced Computing Center (TACC) [4].

Graph benchmark. We evaluate PeeK on eight directed graphs as shown in Table 1. Rmat21 (R21) [18] is a synthetic graph with realistic degree distributions. Livejournal (LJ) [10] and GAP-twitter (GT) [11] are social networks where vertices are users and edges are the interactions. Wikipedia (WL) [46] contains Wikipedia articles as vertices and wikilinks as directed edges. GAP-web (GW) [11] is a web crawl of the .sk domain. Note that R21, LJ, and WL have no edge weights on their original graphs. Therefore, we assign two types of weights, one is a random floating number following normal distributions in the range of (0, 1], and the other is 1 for R21U, LJU, and WLU, respectively. For each graph benchmark, we randomly select 32 pairs of source and reachable target vertices. We use the same source and target pairs for PeeK and compared works.

Table 2: Parallel runtime (s). Hyphen (-) denotes the test cannot complete within 1 hour. The best performance of each graph is highlighted. The speedup of PeeK over current best is in parentheses. The column of × denotes average speedup.

		R21	R21U	LJ	LJU	WL	WLU	GW	GT	×
K=8	Yen	5.3	4.6	13.7	5.8	36.8	24.4	105.7	239.3	6.8
	NC	11.1	1.8	34	9.9	113.6	23.2	247.2	672.4	14.3
	OptYen	3.6	1.9	11.3	4.2	34	19.6	82.9	186.3	5.1
	PeeK	0.7	0.7	1.7	1.6	6.4	5.3	13	22.8	
		(5.1)	*(2.7)*	*(6.5)*	*(2.7)*	*(5.3)*	*(3.7)*	*(6.4)*	*(8.2)*	
K=128	Yen	68.3	61.2	196	44.1	337.1	93.6	174.8	2,168	56.7
	NC	188.6	10.6	665	192	2,405	204.8	-	-	170
	OptYen	39.6	7.8	150	10.8	272.4	24.9	132.4	1,131	28.8
	PeeK	0.8	0.7	2.5	1.6	6.9	5.4	13.6	23.1	
		(49.4)	*(11.5)*	*(60.8)*	*(6.6)*	*(39.2)*	*(4.7)*	*(9.7)*	*(48.9)*	

Table 3: Serial runtime (s). Hyphen (-) denotes the test cannot complete within 1 hour. The best performance of each graph is highlighted. The speedup of PeeK over current best is in parentheses. The column of × denotes average speedup.

		R21	R21U	LJ	LJU	WL	WLU	GW	GT	×
K=8	Yen	30.6	19.4	100.6	12.1	203	44.4	158.4	1,994	7.7
	NC	24.6	9.1	78	19.1	347.1	45	639.9	1,292	7.8
	OptYen	7.4	3.8	39	5.9	68.9	23	113.3	224.3	2.2
	SB	7.7	4.5	30.3	10.3	79.6	31.6	174.5	231.3	2.5
	SB*	7.7	4.4	17.8	8.4	69.6	29.4	196	223.7	2.2
	PeeK	3.5	3.1	7	5.6	26.3	18.6	56.5	115.4	
		(2.1)	*(1.2)*	*(2.5)*	*(1.05)*	*(2.6)*	*(1.2)*	*(2)*	*(1.9)*	
K=128	Yen	576.9	354	2,109	159.5	2,813	254.3	758.3	-	105.9
	NC	421.6	120	1,791	215.1	-	1,238.6	-	-	104
	OptYen	44.8	8.7	353	14.2	347.4	29.3	187.9	-	12.4
	SB	13.3	4.7	108.8	10.6	197.4	31.5	535.0	328.3	5.5
	SB*	9.3	4.6	31.5	8.8	86.8	30.5	430.9	291.3	3.1
	PeeK	3.5	3.1	7	5.7	26.5	18.6	58.3	118.1	
		(2.6)	*(1.5)*	*(4.5)*	*(1.5)*	*(3.3)*	*(1.6)*	*(3.2)*	*(2.5)*	

7.2 Parallel Performance Comparison

This experiment compares the parallel performance of PeeK with OptYen, NC, and Yen. OptYen is the state-of-the-art parallel computation method [5], NC is the node classification algorithm [25], and Yen is the classical Yen's algorithm [70]. They all use the Δ-stepping algorithm for SSSP and implement the same two-level parallel strategies [5] by both concurrently running multiple SSSPs and parallelizing each Δ-Stepping algorithm. We obtained the source codes of Yen, NC, and OptYen from the authors of OptYen [5].

Table 2 summarizes the results of all the tested graphs for two K values, i.e., a small value of 8, and a large value of 128. We use both real runtime and speedup, which is calculated by the runtime of other methods over PeeK. All the methods are running with 32 threads on the same machine. From that, we get three observations.

(i) PeeK significantly outperforms the compared methods on all the tested graphs for both $K = 8$ and 128. In particular, when $K = 8$, PeeK achieves 5.1×, 14.3×, and 6.8× speedup over OptYen, NC, and Yen, respectively, on the average of all the graphs.

(ii) PeeK achieves more speedup for larger K values. That is, when $K = 128$, PeeK achieves 28.8×, 170×, and 56.7× speedup over OptYen, NC, and Yen, respectively. This is mainly due to our K upper bound pruning technique, as the final KSP computation is on the pruned graph, while other methods use the original graph.

(iii) NC performs worst for most of the graphs, especially for large graphs with large K values. This is mainly caused by the dynamic update of the reverse shortest path tree, which is not a good fit for parallel computation. Interestingly, OptYen's simple strategy of only using a static reverse shortest path tree works much better in parallel settings. In particular, OptYen achieves 2.8× and 5.9× speedup over NC for $k = 8, 128$, respectively.

7.3 Serial Performance Comparison

This experiment compares the performance of PeeK with five algorithms for serial execution using one thread. Besides the three algorithms tested in parallel, i.e., OptYen, NC, and Yen, we also compare PeeK with two other algorithms, i.e., SB and SB*. SB is the *Sidetracks-based (SB)* algorithm [47]. SB* algorithm further improves the time efficiency of the SB algorithm by reusing the previously computed SSSP tree [6, 7], while it takes additional space. We obtained the source codes from the authors of SB*. We did not compare SB and SB* for parallel computation because we cannot find any available implementations. We tried to implement by ourselves but face the technical challenge of parallelizing a resumable SSSP algorithm they used, which requires stopping at a certain status and resuming when certain conditions are satisfied.

Table 3 summarizes the serial execution performance when $K = 8, 128$, respectively. We can get three observations. (i) PeeK outperforms all the other algorithms for serial execution. When $K = 8$, it achieves 2.2×, 2.5×, and 2.2× speedup over SB*, SB, and OptYen, respectively. It achieves more than 7× speedup over Yen and NC algorithms. (ii) PeeK outperforms even more when K becomes large. That is, when $K = 128$, PeeK achieves 3.1×, 5.5×, and 12.4× speedup over SB*, SB, and OptYen, respectively. It achieves over 100× speedup over Yen and NC. (iii) SB* is faster than OptYen, especially for large K values. That is, when $K = 128$, SB* achieves 4× speedup over OptYen. The SB* algorithm avoids computing an SSSP from scratch by reusing part of the previously computed SSSP. However, when K is small, e.g., $K = 8$, their runtime is close.

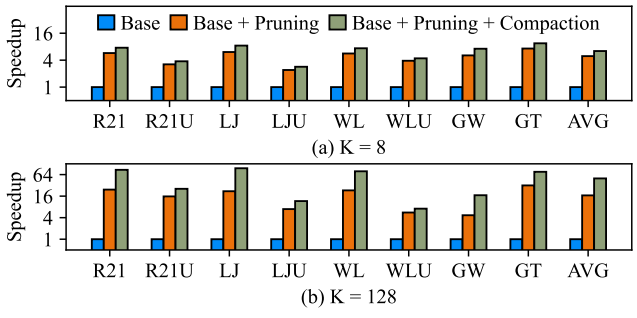


Figure 8: Benefits of the proposed techniques for $K = 8, 128$. The Y-axis is in the log scale.

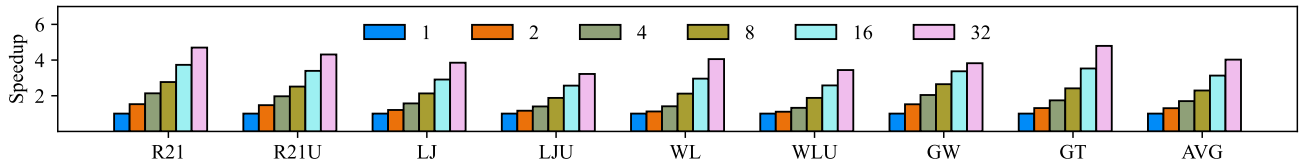


Figure 9: The scalability against thread count on a shared-memory system when $K = 8$.

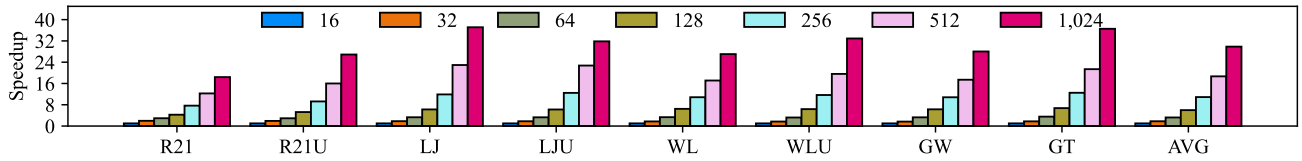


Figure 10: The scalability against core count (16 cores per computing node) on a distributed-memory system when $K = 8$.

7.4 Technique Benefits

This experiment studies the benefits of the proposed techniques for parallel computation, i.e., K upper bound pruning, and adaptive graph compaction. In particular, we use a baseline of PeeK having neither K upper bound pruning nor adaptive graph compaction.

Figure 8 shows the relative speedup compared with the baseline for $K = 8, 128$. (i) Compared with the baseline, the K upper bound pruning technique achieves 4.9× and 16.8× speedup for $K = 8, 128$, respectively. Here, we use the original CSR with status arrays to mark whether a vertex or edge has been pruned. Therefore, the pruned vertices and edges are still in the original CSR, which can lead to redundant computation especially when the kept graph is relatively small. (ii) The adaptive graph compaction technique addresses the issue from the status array. In particular, on top of the K upper bound pruning implementation, the adaptive graph compaction technique adds another 1.5× and 33× speedup for $K = 8, 128$, respectively. Together, both techniques can deliever 6.4× and 50× speedup for $K = 8, 128$, respectively.

7.5 Scalability

Shared-memory scalability. This experiment studies the scalability of PeeK against different thread counts in a shared-memory system. Figure 9 shows the runtime speedup of different numbers of threads over one thread for all the graphs and on average when $K = 8$. One can observe that, PeeK shows a stable speedup increase when the thread count also increases. In particular, compared with one thread, PeeK with 32 threads achieves 4× speedup on average. It achieves the highest speedup, 4.8× for graph GT, which is a large graph with over 1 billion edges.

Distributed scalability. This experiment studies the scalability of PeeK against different numbers of computing nodes in a distributed-memory system. We did not compare with other methods as we were not able to find any available implementations for distributed KSP. We use the number of giga-traversed edges per second (GTEPS) as the metric for runtime, which measures both communication capability and computational power. We scale PeeK from 1 computing node to 64 and use 16 cores per computing node. Figure 10 shows the speedup of using different numbers of cores over 16 cores. One can see, the performance of PeeK stably improves with the increase of computing nodes and cores. Compared with 1 computing node (16 cores), PeeK achieves 30× speedup when it is scaled to 64 computing nodes (1,024 cores) on average. Particularly, with 1,024 cores, PeeK achieves 3.4 GTEPS on average.

7.6 Performance with Different K Values

This experiment studies the performance of different K values. Figure 11 shows the performance change when K increases from 2 to 128. One can observe that PeeK outperforms the compared methods for different K values. More importantly, the larger the K value, the larger the performance gap becomes. This applies to all the methods on the tested graphs as implied by the time complexity $O(Kn(m + n \log n))$. However, PeeK shows a much smaller increasing rate compared with other methods mainly due to the strong pruning power from the K upper bound pruning. In particular, when increasing K from 2 to 128 (i.e., 64×), the runtime of PeeK increases by only 1.1×, while OptYen, NC, and Yen increase by 10.3×, 60.7×, and 18×, respectively. NC shows a dramatic increase rate

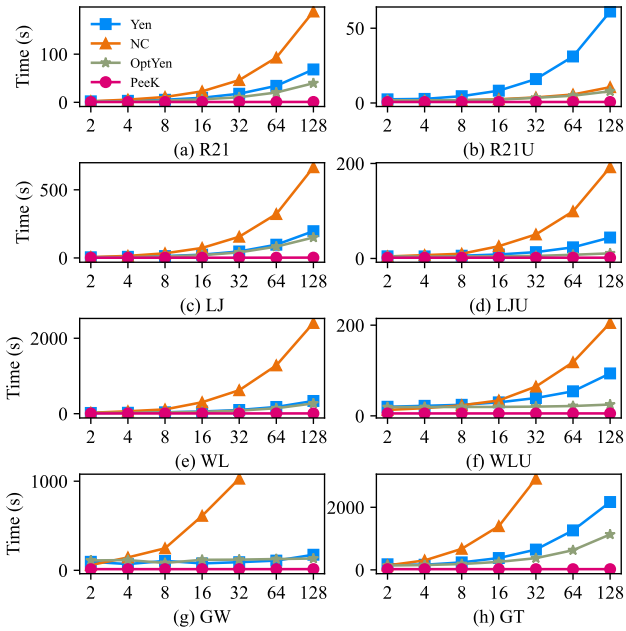


Figure 11: Runtime (s) of different methods on various K values from 2 to 128. The x-axis is in log scale.

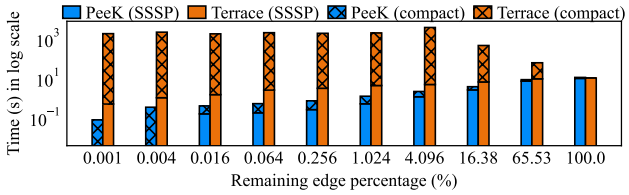


Figure 12: The end-to-end runtime (s) of our adaptive graph compaction method vs. Terrace against the change of the remaining graph size for the Twitter (GT) graph.

mainly due to the high overhead caused by dynamically updating the reverse shortest path tree and vertex colors.

7.7 Comparison with Dynamic Graph Representation

This experiment studies the difference between our adaptive graph compaction method and the dynamic graph update method. We test one of the state-of-the-art dynamic graph update systems, i.e., Terrace [59], which can efficiently deal with the vertex/edge update (insertion/deletion) for streaming graphs in parallel. It uses a hierarchical data structure to store a vertex's neighbors in different data structures depending on the degree of the vertex. We measure the end-to-end runtime performance against different numbers of deleted vertices and edges. Here, we use the runtime of graph update and downstream computation task SSSP as the end-to-end performance. We use SSSP because (i) SSSP is one of the downstream computation tasks after graph update, (ii) Terrace does not support KSP computation. We test on a Twitter (GT) graph. We evaluate a different number of deleted edges by starting from only keeping 0.001% percentage and increasing by 4 times.

Figure 12 shows the end-to-end performance. We can conclude with three interesting observations. (i) The SSSP computation from our implementation and Terrace are comparable. When all the edges are kept (100%), the SSSP computation dominates the runtime. From Figure 12, one can see their runtime is relatively the same. (ii) PeeK outperforms Terrace significantly for end-to-end performance. When only 0.001% vertices are kept, PeeK achieves 23,129× speedup over Terrace. As the graph update for KSP computation often requires pruning the majority of vertices/edges (discussed in Section 4), we select the adaptive graph compaction method instead of the dynamic graph update. (iii) However, we do notice the performance of Terrace dramatically improves when the graph update is fewer. One can see, when the number of remaining edges increases to 65.53%, the speedup of PeeK over Terrace drops to 7×.

8 RELATED WORK

We have discussed Yen's algorithm [70], node classification (NC) [25], OptYen [5], Sidetracks-based (SB) [47], and parallel strategies [5, 63] throughout the paper. This section will discuss other related works.

Postponed NC (PNC) [7] is a variant of NC algorithm. It tries to further reduce the number of SSSP calls observing many of the paths obtained from an expensive SSSP call may not become a final candidate for the K shortest paths. To avoid them, the PNC algorithm temporally puts a non-simple candidate path for each deviation vertex into the candidate set. When this non-simple candidate path is finally extracted, it will "repair" it by running SSSP. Further, the authors of PNC further propose PNC* [7]. It computes the SSSP algorithm on the subgraph only with the yellow vertices instead of the whole graph in the PNC algorithm.

SB* algorithm improves the time efficiency of the SB algorithm by reusing the previously computed SSSP tree [6, 7]. In particular, to compute a new reverse SSSP tree T_{i+1}, it creates a copy of T_i, and recovers T_i for the purpose of T_{i+1} by removing the different vertices and edges. The previously used SSSP algorithm can continue the computation from the recovered tree of T_i. This reduces the duplicate SSSP computation for the same vertices and edges.

Parsimonious Sidetrack-based (PSB) algorithm and two variants PSB-v2, PSB-v3 are proposed to reduce the large memory usage of the SB algorithm [6, 7]. The key idea of PSB is to only store a computed reverse SSSP tree after finding a useful subpath in that tree. PSB-v2 defines a static threshold with the hope of predicting whether a reverse SSSP tree will lead to a path that can become one of the extracted candidates. With that, it only stores the SSSP tree satisfying the threshold. PSB-v3 goes further by dynamically changing the threshold during KSP computation.

Lawler proposed an optimization for Yen's algorithm [49] by reducing the number of SSSPs. Hershberger *et al.* improved the time with $\Theta(n)$ speedup, while it is not stable as replacement paths can fail [34]. Gotthilf and Lewenstein improved the time complexity to $O(Kn(n \log\log n + m))$ [30]. However, it requires the best all-pairs shortest path (APSP) algorithm with $O(mn + n^2 \log\log n)$ using Thorup's component tree [65], which has not been implemented for directed graph [25]. Sedeno-Noda [60] proposed a different solution from Yen's algorithm, but with the same time complexity.

Further, the idea of pruning unnecessary vertices and edges for improving the performance can potentially benefit other graph algorithms, e.g., connectivity algorithms [38, 43], centrality algorithms [44], graph embedding algorithms [14, 19], and graph neural networks [27, 32]. We would like to explore them in the future.

9 CONCLUSION

This paper devises PeeK, a new method for K shortest path computation with two new techniques, i.e., K upper bound pruning, and adaptive graph compaction. We also design efficient techniques to parallelize and distribute PeeK. Compared with five other algorithms, PeeK achieves 2.2× and 5.1× over state-of-the-art serial and parallel methods, respectively, when $K = 8$. More importantly, the runtime of PeeK is barely affected by the increase of K. That is, when K increases from 2 to 128 (64×), PeeK only increases 1.1×, while the state-of-the-art method increases 10.3×.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable suggestions. We would also like to express our grateful thanks to the authors of OptYen for sharing the source code with us. This work was supported in part by National Science Foundation grants 2319975, 2331301, 2212370, 2331536, and 2326141. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views of the National Science Foundation, or the U.S. Government.

REFERENCES

[1] [n. d.]. Block indirect sort algorithm. https://www.boost.org/doc/libs/1_81_0/libs/sort/doc/html/sort/parallel.html#sort.parallel.block_indirect_sort.

[2] [n. d.]. Distributed Sorting Algorithms. https://brunomaga.github.io/Distributed-Sort.

[3] [n. d.]. Graph500: reference implementations. https://github.com/graph500/graph500.

[4] [n. d.]. Texas Advanced Computing Center. https://www.tacc.utexas.edu.

[5] Deepak Ajwani, Erika Duriakova, Neil Hurley, Ulrich Meyer, and Alexander Schickedanz. 2018. An Empirical Comparison of K-Shortest Simple Path Algorithms on Multicores. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*. 1–12.

[6] Ali Al Zoobi, David Coudert, and Nicolas Nisse. 2020. Space and time trade-off for the k shortest simple paths problem. In *SEA 2020-18th International Symposium on Experimental Algorithms*, Vol. 160. 13.

[7] Ali Al Zoobi, David Coudert, and Nicolas Nisse. 2021. *Finding the k Shortest Simple Paths: Time and Space trade-offs.* Ph. D. Dissertation. Inria; I3S, Université Côte d'Azur.

[8] M Albulet. 2016. Spacex non-geostationary satellite system: Attachment a technical information to supplement schedules. *US Fed. Commun. Comm., Washington, DC, USA, Rep. SAT-OA-20161115-00118* (2016).

[9] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.

[10] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.

[11] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[12] E Blelloch Guy. 1990. *Prefix Sums and Their Applications.* Technical Report. Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

[13] Mumeng Bo and Meihui Zhang. 2021. Learning Dynamic Coherence with Graph Attention Network for Biomedical Entity Linking. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[14] Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang. 2020. Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[15] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[16] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, seattle, Washington USA, 1–12.

[17] Jian Cao, Qiang Li, Yuede Ji, Yukun He, and Dong Guo. 2016. Detection of forwarding-based malicious URLs in online social networks. *International Journal of Parallel Programming* 44, 1 (2016), 163–180.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[19] Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. 2023. API2Vec: Learning Representations of API Sequences for Malware Detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–273.

[20] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[21] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 918–934.

[22] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 287–290.

[23] David Eppstein. 1998. Finding the k shortest paths. *SIAM Journal on computing* 28, 2 (1998), 652–673.

[24] Bo Feng, Qiang Li, Yuede Ji, Dong Guo, and Xiangyu Meng. 2019. Stopping the cyberattack in the early stage: assessing the security risks of social network users. *Security and Communication Networks* 2019 (2019).

[25] Gang Feng. 2014. Finding k Shortest Simple Paths in Directed Graphs: A Node Classification Algorithm. *Networks* 64, 1 (2014), 6–17.

[26] Rod Fleck. 2019. Application of Kuiper Systems LLC for Authority to Launch and Operate a Non-Geostationary Satellite Orbit System in Ka-band Frequencies.

[27] Qiang Fu, Yuede Ji, and H Howie Huang. 2022. TLPGNN: A lightweight two-level parallelism paradigm for graph neural network computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 122–134.

[28] Giorgio Gallo and Stefano Pallottino. 1988. Shortest path algorithms. *Annals of operations research* 13, 1 (1988), 1–79.

[29] Giacomo Giuliari, Tommaso Ciussani, Adrian Perrig, and Ankit Singla. 2021. {ICARUS}: Attacking low earth orbit satellite networks. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 317–331.

[30] Zvi Gotthilf and Moshe Lewenstein. 2009. Improved Algorithms for the K Simple Shortest Paths and the Replacement Paths Problems. *Inform. Process. Lett.* 109, 7 (2009), 352–355.

[31] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.

[32] Haoyu He, Yuede Ji, and H Howie Huang. 2022. Illuminati: Towards explaining graph neural networks for cybersecurity analysis. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 74–89.

[33] Yukun He, Qiang Li, Jian Cao, Yuede Ji, and Dong Guo. 2017. Understanding socialbot behavior on end hosts. *International Journal of Distributed Sensor Networks* 13, 2 (2017), 1550147717694170.

[34] John Hershberger, Matthew Maxel, and Subhash Suri. 2007. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)* 3, 4 (2007), 45–es.

[35] Daniel S. Hirschberg. 1978. Fast parallel sorting algorithms. *Commun. ACM* 21, 8 (1978), 657–661.

[36] Walter Hoffman and Richard Pavley. 1959. A method for the solution of the n th best path problem. *Journal of the ACM (JACM)* 6, 4 (1959), 506–514.

[37] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *16th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*.

[38] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *16th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*.

[39] Yuede Ji, Lei Cui, and H Howie Huang. 2021. Vestige: Identifying Binary Code Provenance for Vulnerability Detection. In *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 287–310.

[40] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. 2021. DEFInit: An Analysis of Exposed Android Init Routines. In *30th USENIX Security Symposium (USENIX Security)*.

[41] Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. 2016. Combating the evasion mechanisms of social bots. *Computers & Security* (2016).

[42] Yuede Ji, Yukun He, Xinyang Jiang, and Qiang Li. 2014. Towards social botnet behavior detecting in the end host. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 320–327.

[43] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 731–742.

[44] Yuede Ji, Hang Liu, and H. Howie Huang. 2020. SwarmGraph: Analyzing Large-Scale In-Memory Graphs on GPUs. In *International Conference on High Performance Computing and Communications (HPCC)*. IEEE.

[45] Masahiko Jinno, Bartlomiej Kozicki, Hidehiko Takara, Atsushi Watanabe, Yoshiaki Sone, Takafumi Tanaka, and Akira Hirano. 2010. Distance-adaptive spectrum resource allocation in spectrum-sliced elastic optical path network [topics in optical communications]. *IEEE Communications Magazine* 48, 8 (2010), 138–145.

[46] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[47] Denis Kurz and Petra Mutzel. 2016. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. *27th International Symposium on Algorithms and Computation* (2016).

[48] Dominique LaSalle, Md Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. 2015. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 14.

[49] Eugene L Lawler. 1972. A Procedure for Computing the K Best Solutions to Discrete Optimization Problems and Its Application to the Shortest Path Problem. *Management Science* 18, 7 (1972), 401–405.

[50] John Lhota and Lei Xie. 2016. Protein-fold recognition using an improved single-source K diverse shortest paths algorithm. *Proteins: Structure, Function, and Bioinformatics* 84, 4 (2016), 467–472.

[51] Bi-Qing Li, Tao Huang, Lei Liu, Yu-Dong Cai, and Kuo-Chen Chou. 2012. Identification of colorectal cancer related genes with mRMR and shortest path in protein-protein interaction network. *PloS one* 7, 4 (2012), e33393.

[52] Bo Liu, Gao Cong, Yifeng Zeng, Dong Xu, and Yeow Meng Chee. 2013. Influence spreading path and its application to the time constrained social influence maximization problem and beyond. *IEEE Transactions on Knowledge and Data Engineering* 26, 8 (2013), 1904–1917.

[53] Rui Liu, Raj Velamur Srinivasan, Kiyana Zolfaghar, Si-Chi Chin, Senjuti Basu Roy, Aftab Hasan, and David Hazel. 2014. Pathway-finder: An interactive recommender system for supporting personalized care pathways. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 1219–1222.

[54] Tiantian Liu, Zijin Feng, Huan Li, Hua Lu, Muhammad Aamir Cheema, Hong Cheng, and Jianliang Xu. 2020. Shortest path queries for indoor venues with temporal variations. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2014–2017.

[55] Nirmesh Malviya, Samuel Madden, and Arnab Bhattacharya. 2011. A continuous query system for dynamic route planning. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 792–803.

[56] Ulrich Meyer and Peter Sanders. 2003. Δ-Stepping: A Parallel Single Source Shortest Path Algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[57] Sara Nazari, M Reza Meybodi, M Ali Salehigh, and Sara Taghipour. 2008. An advanced algorithm for finding shortest path in car navigation system. In *2008 First International Conference on Intelligent Networks and Intelligent Systems*. IEEE, 671–674.

[58] Uyen TV Nguyen, Shanika Karunasekera, Lars Kulik, Egemen Tanin, Rui Zhang, Haolan Zhang, Hairuo Xie, and Kotagiri Ramamohanarao. 2015. A randomized path routing algorithm for decentralized route allocation in transportation networks. In *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. 15–20.

[59] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1372–1385.

[60] Antonio Sedeño-Noda. 2012. An Efficient Time and Space K Point-to-Point Shortest Simple Paths Algorithm. *Appl. Math. Comput.* 218, 20 (2012), 10244–10257.

[61] Robert Sedgewick. 2001. *Algorithms in C, part 5: graph algorithms*. Pearson Education.

[62] Yu-Keng Shih and Srinivasan Parthasarathy. 2012. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics* 28, 12 (2012), i49–i58.

[63] Avadhesh Pratap Singh and Dhirendra Pratap Singh. 2015. Implementation of K-shortest Path Algorithm in GPU Using CUDA. *Procedia Computer Science* 48 (2015), 5–13.

[64] Edgar Solomonik and Laxmikant V Kale. 2010. Highly scalable parallel sorting. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[65] Mikkel Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)* 46, 3 (1999), 362–394.

[66] Xin Wan, Lei Wang, Nan Hua, Hanyi Zhang, and Xiaoping Zheng. 2011. Dynamic routing and spectrum assignment in flexible optical path networks. In *2011 Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference*. IEEE, 1–3.

[67] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[68] Brian Wheatman and Helen Xu. 2021. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 31–45.

[69] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 590–604.

[70] Jin Y Yen. 1971. Finding the K Shortest Loopless Paths in a Network. *Management Science* 17, 11 (1971), 712–716.

[71] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. Gman: A graph multi-attention network for traffic prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1234–1241.

[72] Yunhui Zheng and Xiangyu Zhang. 2013. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 652–661.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

The main contributions of this paper are three-fold. (i) We propose a K upper bound pruning technique to prune the vertices and edges that will not appear in any of the K shortest paths. In the artifact, we first run the single source shortest path (SSSP) algorithm from the source vertex to get the shortest paths from the source to all the other vertices. Second, we run another SSSP to get the shortest paths from all the other vertices to the target vertex. Finally, we identify the K shortest valid paths.

(ii) We propose an adaptive graph compaction method between edge swap and graph regeneration-based compaction methods. In the artifact, edge swap-based graph compaction will modify the original Compressed Sparse Row (CSR) by moving the deleted neighbors of a vertex to the end so that the kept neighbors are consecutive. Graph regeneration will produce a new CSR.

(iii) We parallelize the proposed method by classifying all the jobs as data parallel, embarrassingly parallel, or task parallel jobs. In the artifact, K upper bound pruning is mostly designed as a data parallel job, the adaptive graph compaction is an embarrassingly parallel job, and the KSP computation is a task parallel job.

REPRODUCIBILITY OF EXPERIMENTS

Table 1: The figures and tables that can be reproduced from this artifact.

Figure/ Table	Description	Estimated runtime (h)
Figure 1	Percentage of the covered vertices and edges against different K values for a Twitter (GT) graph.	6
Figure 4	Percentage of pruned vertex and edge by K upper bound pruning for eight graphs when K is 8 and 128.	1
Figure 6	The end-to-end performance of graph regeneration, edge swap, and status array-based methods plus the downstream KSP computation on the Twitter graph.	6
Figure 8	Benefits of the proposed techniques for K = 8, 128 with 32 threads.	3
Figure 9	Runtime (s) of different methods on various K values from 2 to 128 and 32 threads.	3
Figure 10	The scalability against thread count on a shared-memory system when K = 8.	4
Figure 11	The scalability against the number of cores on a distributed-memory system when K = 8.	4
Table 2	Parallel runtime for 32 threads.	1
Table 3	Serial runtime.	4

We implemented a prototype, named PeeK. In this artifact, we compile it with GCC 8.5.0, O3 optimization level. We use OpenMP 4.5 as the multithreading library, OpenMPI as the message-passing interface library, and Boost 1.81.0 as the sorting library. The single-machine experiments are performed on a server with two Intel Xeon Silver 4309Y CPUs running Rocky Linux 8.6 and 512GB memory. For distributed experiments, we run them on the servers of the Texas Advanced Computing Center (TACC).

We evaluate PeeK on eight directed graphs and randomly select 32 pairs of source and reachable target vertices. We use the same source and target pairs for PeeK and the compared works.

Table 1 summarizes the figures and tables used in our paper with the estimated runtime. Particularly, the paper comes with 11 figures and 3 tables, while Figure 2, Figure 3, Figure 5, Figure 7, and Table 1 are not related to the experiments.

ARTIFACT DEPENDENCIES REQUIREMENTS

None

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

None