Eppstein's Algorithm

Rishik Balerao, Utkarsh Goyal, Jacob Ezieme, Vrishak Vemuri

April 2026

1 Introduction

For many path-finding applications, it is useful to find multiple optimal paths, not just the shortest one. We see this in Google Maps, for example, which allows the user to pick between multiple paths to their destination. An efficient algorithm to calculate the k shortest paths from a source node to a target node would help scale capabilities like this. A variant of Dijkstra's, as described later in this document, can be used to solve this problem at a cost of $O(km \log km)$ runtime, where m is the number of edges. For dense graphs with many edges, which are common in real-world applications, this performance is not ideal because the additional cost of an extra path scales with the number of edges in the graph. We'd prefer to have a solution where the extra cost of getting the $k + 1^{th}$ shortest path is minimized. In particular, we have a target runtime of $O(m \log m + k \log k)$.

Our problem statement is as follows. Given an arbitrary weighted directed graph, we wish to find the k shortest paths from a source node s to a target node t. We assume that every edge has non-negative weight, specifically edges with 0 weight are allowed. We also allow for the graph to have cycles and repeated edges.

2 Overall Intuitive Approach

Eppstein's Algorithm is best explained incrementally by optimizing the naive solution. As such, we have structured this section to slowly build up to the final algorithm.

2.1 Naive Approach

Our first attempt is a modified version of Dijkstra's Algorithm that computes the k shortest path lengths from a source node s to a target node t.

Algorithm 1 Modified Dijkstra's Algorithm

Require: Graph (V, E), source s, target t, integer k
Ensure: k shortest path lengths from s to t

```
pq ← ∅
result ← ∅
count[v] ← 0   ∀v ∈ V
INSERT(pq, (0, s))
while pq ≠ ∅ do
    (d, u) ← EXTRACTMIN(pq)
    if count[u] ≥ k then
        continue
    end if
    count[u] ← count[u] + 1
    if u = t then
        APPEND(result, d)
        continue
    end if
    for all (u, v, w) ∈ E(u) do
        INSERT(pq, (d + w, v))
    end for
end while
return result
```

This modification introduces an additional array *count* that tracks how many times each node has been visited. Because edge weights are guaranteed to be positive, the first k times we visit a node v will be from the k shortest paths from s to v. Once a node v has been visited k times, we will no longer visit v because the $k + 1^{th}$ shortest path to v cannot be a part of any of the k shortest paths from s to t. Because this algorithm visits each node up to k times, it will also add each outgoing edge up to k times, giving us a final complexity of $O(km \log km)$, which comes from the cost of adding km edges to the priority queue.

2.2 First Optimization

To progress towards a more optimal solution, we wish to characterize the extra "cost" incurred by choosing one edge over another. To do so, we let

$$sidetrack(u, v, w) := d_v + w - d_u,$$

where d_u is the cost of the shortest path from u to t. This represents the extra cost we incur by choosing the edge (u, v, w) at node u instead of the optimal next edge from u, assuming we move optimally otherwise. By definition, $sidetrack(u, v, w) \geq 0$ for all $(u, v, w) \in E$.

In addition, for any node u let $nxt(u)$ be the optimal next node to visit, given we are currently at u, with ties broken arbitrarily. Denote by G' a transformed graph which contains all of the same nodes as G. For each edge (u, v, w) in E, we create edge $(u, v, sidetrack(u, v, w))$ in E', the set of all edges in G'. We then remove all edges $(u, nxt(u), \cdot)$, so that only sidetrack edges for non-optimal edges remain. Additionally, for each node u, we add all sidetrack edges $(nxt(u), \cdot, \cdot)$, $(nxt(nxt(u)), \cdot, \cdot)$, ... to the list of sidetrack edges from u. The following algorithm summarizes this procedure.

Algorithm 2 BUILD-SIDETRACK-GRAPH($N, E, nxt, sidetrack$)

Require: Nodes N, Edges E, shortest-path successor function nxt, sidetrack weight function $sidetrack$
Ensure: Transformed graph edge set E', and sidetrack edge lists $S[u]$ for all $u \in N$

```
E' ← ∅
for each edge (u, v, w) ∈ E do
    E' ← E' ∪ {(u, v, sidetrack(u, v, w))}
end for
for each node u ∈ N do
    if nxt(u) ≠ NIL then
        Remove one occurrence of (u, nxt(u), 0) from E'
    end if
end for
for each node u ∈ N do
    S[u] ← all sidetrack edges from u and its shortest-path successors
end for
return E', S
```

An example graph is shown below. Figure 1 shows the initial state with just the edge weights and the distances to t, while Figure 3 shows the new graph G' after applying the transformation.

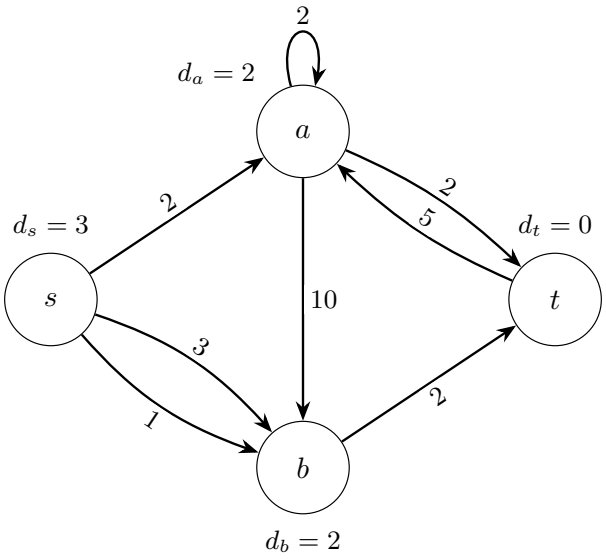

Figure 1: Original Graph G

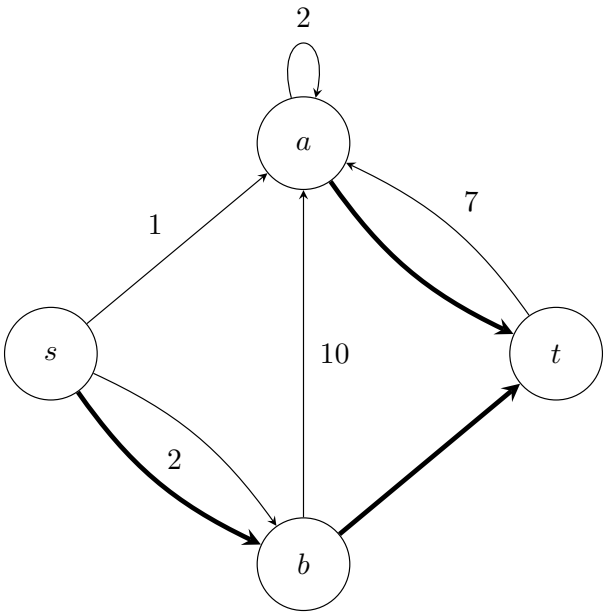

Figure 2: Transformed Graph G'. Nxt edges are in bold

There is a bijection between paths from s to t in G and paths from s to anywhere in G'. To convert a path in G' into a regular path in G, take the sequence of sidetrack edges and connect them by following the optimal shortest-path tree from the head of one sidetrack to the tail of the next, ultimately following the tree to t. Because any valid s-t path in G uniquely diverges from the optimal tree via a specific sequence of non-tree edges, this process can be perfectly inverted to extract the original sequence and get the old path back. Thus, mapping the paths between the two graphs creates a bijection. Moreover, it can be shown that a path of weight d in G' corresponds to a path of weight $d + d_s$ in G, which means that finding the k shortest paths from s to anywhere in G' is the same as finding the k shortest paths from s to t in G.

Thus, to get the k shortest path lengths in G, we just need to get the k shortest paths in G'. The following algorithm summarizes a k-pop algorithm that accomplishes this.

Algorithm 3 EPPSTEIN-K-POP(G', s, d, k)

Require: Transformed graph G' (containing sidetrack lists S), source node s, shortest-path distance array d, integer k
Ensure: The lengths of the k-shortest paths in the original graph
  $Q \leftarrow$ empty min-heap
  PUSH$(Q, (d[s], s))$ ▷ Initialize with the optimal path length
  for $i \leftarrow 1$ to k **do**
    $(l, u) \leftarrow$ POP(Q)
    yield path of length l ▷ Found the i-th shortest path
    for each sidetrack edge $(u, v, w) \in S[u]$ in G' **do**
      PUSH$(Q, (l + w, v))$ ▷ Branch into a new valid sidetrack path
    end for
  end for

The runtime of this algorithm is still $\mathcal{O}(km \log km)$ because there can be up to m sidetrack edges out of each node u because of the pre-processing step where we add all parent sidetrack edges to children, so we add up to km items to the priority queue.

2.3 Second Optimization

The previous approach was slow because it added up to m edges to the priority queue at each iteration of the k-pop algorithm. By changing how we store our sidetrack edges for each node, we can drastically speed up this bottleneck. In particular, instead of storing sidetrack edges for each node as a list, we can instead store them in a min-heap. We call this modified graph G". Below, we show G" for our sample.

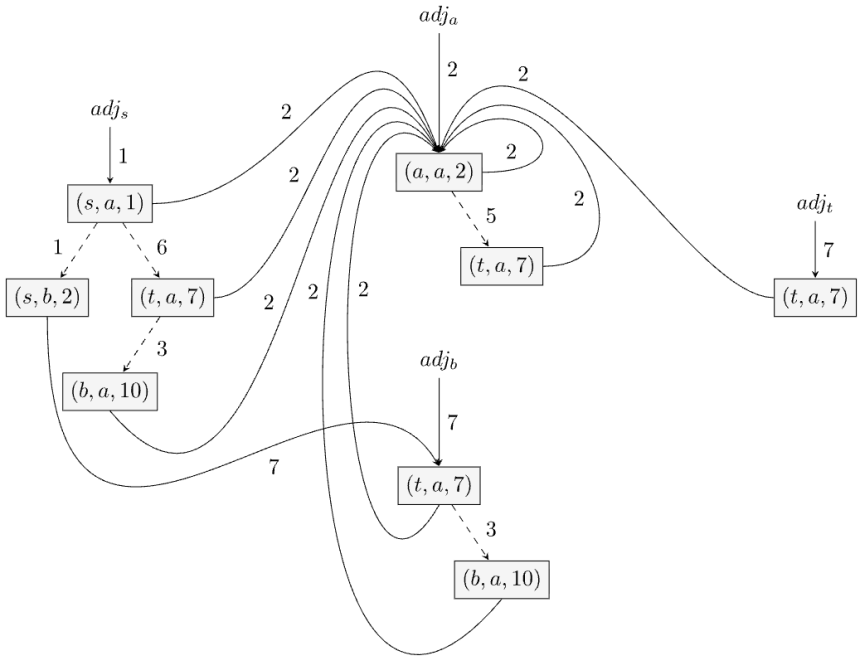

Figure 3: G''

We then define two distinct types of directed connections between these heap nodes. First, a node representing sidetrack edge (u, v, w) points directly to the root of the heap adj_v, as shown with the solid lines above. The cost of this transition is simply the weight of that root edge. Second, each node points to up to two children within its own heap adj_u, as shown with the dashed lines above. The cost of this transition is the

difference in weight between the child edge and the parent edge. This relative weighting scheme ensures that the accumulated length of a path in G'' exactly matches the total cost of the corresponding sidetrack sequence. Consequently, we can find the k shortest paths simply by executing k-pop on G'' as before. However, since the maximum out-degree of any one node in G'' is 3, our runtime for k-pop becomes $O(k \log k)$, removing the dependence on m for each successive path. To create the sidetrack graph, we need one iteration of Dijkstra's algorithm at a cost of $O(m \log m)$ runtime, giving us a total cost of $O(m \log m + k \log k)$ runtime.

3 Implementation

3.1 Optimal Merging Order

One might wonder how to best implement the merging step in the construction of the sidetrack graph G'. We require that adj_u contains all sidetrack edges from $nxt(u)$, $nxt(nxt(u))$, ... Luckily, we can implement this in a way such that each node u only needs to merge its adjacency list adj_u with that of one other node: $adj_{nxt(u)}$. If we process nodes in reverse order from the target, ensuring that $nxt(u)$ is processed before u for all nodes u, then $nxt(u)$ will already contain all sidetrack edges from $nxt(nxt(u))$, $nxt(nxt(nxt(u)))$ ..., so we only have to merge adj_u with $adj_{nxt(u)}$ to get all sidetrack edges from all of its successors in the optimal path tree. This can be implemented with a BFS on the transpose of the shortest path tree (constructing a graph where there is an edge from $nxt(u)$ to u for all nodes u and starting the traversal at t). Each time a node u is visited, you merge adj_u with $adj_{nxt(u)}$.

3.2 Leftist Heap

The heap optimization for storing adj_u does not come for free. By introducing extra structure to adj_u, we must adjust our procedure for merging adj_u and $adj_{nxt(u)}$ to create the sidetrack graph. In particular, we introduce a variant of a traditional binary heap called a leftist heap. A (min) leftist heap has the following properties:

- each node has up to 2 children
- each parent node has key smaller than that of its children
- each node's right child's null path length is less than its left child's null path length

The null path length of a node is defined as the shortest path from that node to a node missing at least one child. More formally, the null path length of a null reference is -1, and the null path length of any existing node is 1 plus the minimum of the null path lengths of its left and right children. This property ensures that the right spine of the tree is always the shortest path to a null node, allowing us to merge two heaps of size n and m in $O(\log n + \log m)$ time by always traversing and recursively merging along their right spines.

To implement a leftist heap, in each node we store the following information:

- the key for the current node
- a reference or pointer to the left child node
- a reference or pointer to the right child node
- an integer tracking the null path length of the node

After a merge, we enforce the leftist property by swapping left and right children whenever the null path length of the right child exceeds that of the left child. Additionally, after merging heaps A and B, our algorithm will still want to access A and B individually. In other words, the act of merging adj_u and $adj_{nxt(u)}$ should not modify $adj_{nxt(u)}$. To ensure this, we make shallow copies of our nodes whenever we modify their pointers during merges, ensuring that the old nodes with old references are still accessible. As a result of this, adj_u and $adj_{nxt(u)}$ will share most of their nodes after the merge (except for the $\log(n)$ modified in the merge). This is because we only make shallow copies of nodes we modify, maintaining references to

their childen instead of copying entire child sub-heaps. This entire procedure is summarized in the following algorithm:

Algorithm 4 PERSISTENT-MERGE(A, B)

Require: Leftist heaps A and B, where nodes contain key, $left$, $right$, and npl attributes
Ensure: A new persistent leftist heap merging A and B, leaving the original heaps unmodified

```
if A = NIL then
    return B
end if
if B = NIL then
    return A
end if
if A.key > B.key then
    exchange A with B                                  ▷ Ensure A holds the minimum key
end if
C ← ALLOCATE-NODE()                                    ▷ Begin shallow copy optimization
C.key ← A.key
C.left ← A.left                                        ▷ Structurally share the entire left subtree
C.right ← PERSISTENT-MERGE(A.right, B)                 ▷ Merge along the right spine
if C.left = NIL or (C.right ≠ NIL and C.left.npl < C.right.npl) then
    exchange C.left with C.right                       ▷ Swap to maintain the leftist property
end if
if C.right = NIL then
    C.npl ← 0
else
    C.npl ← C.right.npl + 1
end if
return C                                               ▷ Return the new root, preserving prior history
```

4 Programming Challenge

The goal of the programming challenge was to find the longest possible walk on a graph that stays under a fixed budget. The relevant constraints are

$$2 \leq n \leq 10^5, \qquad 0 \leq m \leq 10^5,$$

and the total number of distinct walks from the starting node s to the ending node t is at most 10^5.

The final constraint is particularly important because it is what makes Eppstein's algorithm applicable. This is not immediately obvious if the undergraduate contestant is not careful. The main conceptual twist is that Eppstein's algorithm would not be feasible without this bound on the total number of distinct walks.

Another significant implementation challenge is identifying and printing the optimal path produced by Eppstein's algorithm. This is difficult for two main reasons. First, the total number of nodes across all paths considered by Eppstein's algorithm can exceed 10^9, which would not satisfy the time constraints. Second, the backtracking logic required to reconstruct the path is nontrivial. Therefore, the solution must implement backtracking only for the single path that is ultimately printed.

Although Eppstein's algorithm itself is difficult, the challenge balances implementation and algorithmic reasoning by requiring contestants to print the path. This forces them to understand how edges are tracked throughout Eppstein's algorithm and to recognize that printing the optimal path is possible without increasing the asymptotic time complexity.

The intended solution relies on observing that Eppstein's algorithm is sufficient under the given constraints and then modifying the implementation so that only the printed path is reconstructed. The final time complexity remains the same as a standard Eppstein-style implementation:

$$\Theta(m \log m + k \log k),$$

where k is the number of paths that must be considered.

Because implementing Eppstein's algorithm is already highly challenging, most of the difficulty of this problem is concentrated in understanding why the algorithm satisfies the constraints and in modifying the code to correctly print the optimal path.

5 Key Ideas for Solving the Programming Challenge

The key idea of the programming challenge is to use Eppstein's algorithm to enumerate candidate walks from s to t in increasing order of cost, and then choose the longest walk whose total cost stays under the given budget. The important observation is that the number of distinct s-to-t walks is guaranteed to be at most 10^5. This bound makes it possible to consider the relevant walks without the enumeration becoming too large.

The main challenge is recognizing why this constraint matters. Without the bound on the number of distinct walks, Eppstein's algorithm could generate too many candidates, and the approach would not satisfy the time limits. Therefore, the problem is not only about implementing a known shortest-path enumeration algorithm, but also about understanding why that algorithm is valid under the specific constraints of the problem.

Another key challenge is reconstructing and printing the chosen walk. A naive solution might try to store every full path generated by the algorithm, but the total number of vertices across all generated paths can be extremely large. Instead, the model solution stores only the necessary sidetrack information and reconstructs the final selected walk at the end. This keeps the implementation efficient while still allowing the correct path to be printed.

These changes are interesting because they force contestants to go beyond a direct implementation of Eppstein's algorithm. They need to understand how paths are represented internally, how sidetrack edges define a walk, and why only the final answer should be reconstructed. At the same time, the changes are not too difficult because they do not require inventing a new algorithm. Once the contestant understands the structure of Eppstein's algorithm, the required modification is a natural extension of the path-tracking logic.

The complexity of the model solution is

$$\Theta(m \log m + k \log k),$$

where m is the number of edges and k is the number of candidate walks considered. The $m \log m$ term comes from the shortest-path and heap construction steps used by Eppstein's algorithm, while the $k \log k$ term comes from extracting candidate walks from the priority queue. Since the problem guarantees that the total number of distinct walks from s to t is at most 10^5, this running time is feasible under the given constraints. The path reconstruction step does not increase the asymptotic complexity because it is performed only for the final selected walk.

6 Conclusion

In this paper we present a very fast approach to compute the k-shortest paths in a graph, Eppstein's algorithm. Through a series of optimizations, we ensure that the root node knows which sidetracks to take for the next-smallest cost, and this can be repeated until k shortest paths are achieved. Although each node indiviudally knows the cost of the sidetracks that are adjacent to it, merging all sidetracks to hand over to the root node in $O(n \log n)$ time is done through a leftist heap thanks to its $O(\log n)$ time merge operation. Our programming challenge tries to extend Eppstein's algorithm by asking the student to output the k shortest path to take, under some fixed cost where they must keep incrementing to find k, then print the path through summed sidetracks from the original Djikstra path.