Gavin Crigger, Tao Groves, Matthew Lucio, Justin Park

CS 4501

Professor Floryan

28 April 2025

Eppstein's Algorithm - Executive Summary

Introduction / Problem Statement:

Our project topic is on David Eppstein's 1998 algorithm for finding k shortest paths in a given graph from nodes s to t. Expanding on Dijkstra's algorithm, this routine uses persistent heaps and a unique reconstruction of the input graph to optimize down to the target runtime of $O(m + nlogn + k)$, where m represents number of edges, n represents number of vertices, and k represents number of paths returned. The k-shortest-paths-problem has applications in network routing (e.g., finding alternate routes), scheduling such as critical path computation in PERT charts and many optimization problems solved by dynamic programming or more complicated matrix searching techniques, such as the knapsack problem, sequence alignment in molecular biology, construction of optimal inscribed polygons, and length-limited Huffman coding.

The naive approach takes Dijkstra's algorithm, with a runtime of $O(nlogn)$, and repeats it k times to get an overall runtime of $O(knlog(kn))$. Eppstein's algorithm significantly improves on this approach by only running Dijkstra once, merging leftist heaps for amortized linear time, and reconstructing is linear in path length. In comparison with Yen's algorithm, Eppstein's scales better with larger k and requires less memory $O(kn)$ vs $O(m + n + k)$.

Overall Intuitive Approach / Solution:

A key concept that allows for Eppstein's optimization of this problem is the sidetrack edge. A sidetrack edge (u, v, w) is an edge not on the optimal path from vertex s to t, and its

sidetrack weight is equal to the optimal weight of the path from *u* to *t* plus the original weight of edge (*u*, *v*) minus the optimal weight of the path from *v* to *t*. That is,

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

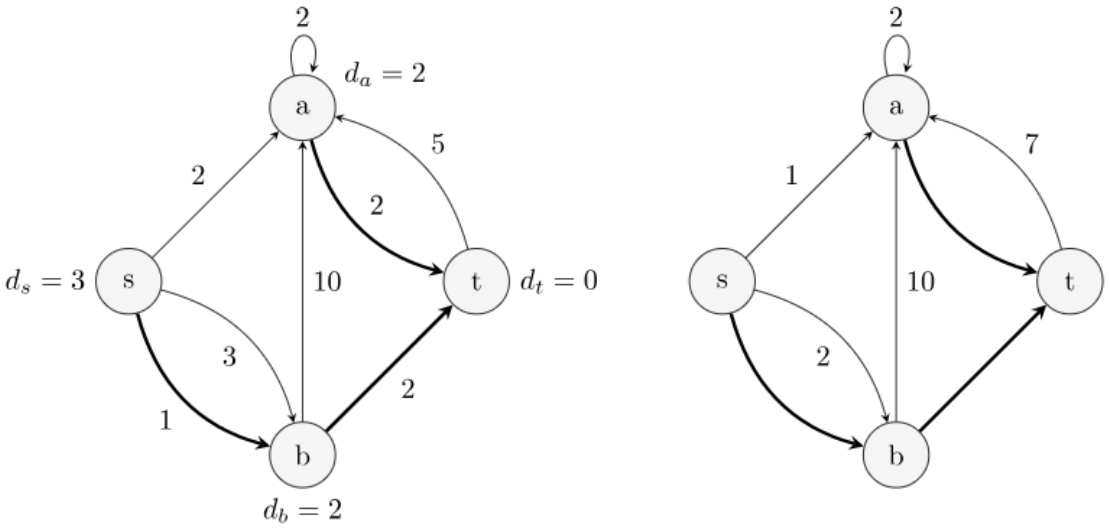

Fig 1. An example graph with d values and the same graph with sidetrack costs.

By using sidetrack edges, Eppstein's algorithm addresses the key issue with the naive approach to the k shortest paths problem that paths must be generated through multiple pops from a heap. A graph of sidetrack edges can query an entire shortest path with one pop, as sidetrack edges can be used alongside the optimal path from *s* to *t* to construct a new path. With this knowledge, the next step in speeding up the approach to this problem is finding a way to best construct a sidetrack graph such that both construction and heap popping is done quickly.

Implementation:

To implement Eppstein's algorithm, we use a combination of Dijkstra's algorithm and a specialized heap structure to efficiently manage sidetrack edges. Dijkstra's algorithm utilizes a priority queue to efficiently manage the vertices being processed.

```
def dijkstra(g, src, n):
    dist = [INF] * n
    parent = [-1] * n
    dist[src] = 0
    heap = [(0, src)]  # (distance, vertex)
    while heap:
        d_u, u = heapq.heappop(heap)
        if d_u != dist[u]:
            continue
        for w, v in g[u]:
            if d_u + w < dist[v]:
                dist[v] = d_u + w
                parent[v] = u
                heapq.heappush(heap, (dist[v], v))
    return dist, parent
```

To do this, we utilize a leftist heap (`EHeap`) to optimally select sidetrack edges. Each node stores the null-path length (NPL), cost difference, target vertex, and pointers to its left and right children. When inserting, we want the left side to be heavier (tree is left-skewed because the right child always has the smaller NPL, so the path you recurse down during merges, the right spine, is kept as short as possible) so that merge operations are faster (amortized *O(log(n))*).

```
class EHeap:
       def __init__(self, rank, key, value, left, right):
       self.rank = rank      # null-path length (NPL) for leftist property
       self.key = key        # cost difference of this sidetrack
       self.value = value    # target vertex of this sidetrack edge
       self.left = left      # left child heap
       self.right = right    # right child heap
def insert(a, k, v):
"""Insert a (key=k, value=v) into heap 'a', returning new root. This is persistent:
original 'a' remains unchanged."""
```

```
        # Base case: empty heap or new key is smaller than root
        if not a or k < a.key:
               return EHeap(1, k, v, a, None)
        # Recursively insert into right subtree
        l, r = a.left, EHeap.insert(a.right, k, v)
        # Ensure leftist property: left.rank >= right.rank
        if not l or (r and r.rank > l.rank):
        l, r = r, l
        # Update rank: 1 + rank of right child
        new_rank = (r.rank + 1) if r else 1
        # Return new heap node preserving original key/value
        return EHeap(new_rank, a.key, a.value, l, r)

def __lt__(self, other):
         # Needed for heapq but actual comparison is by external key
         return False
```

The actual algorithm uses these to compute the k^{th} *shortest path*. It builds a reverse graph to run reverse Dijkstra's and build a shortest path tree to identify edges used/unused in the current path. It then creates a leftist heap for each node consisting of edges not in the shortest path tree to store the additional cost and children for merging. Once these are created, the paths are enumerated to track the total path cost, path taken, and pointer to the heap node representing the alternate path taken.

```
def eppstein_k_shortest_paths(g, src, dst, k, n):
    """
    Compute k shortest path costs from src to dst using Eppstein's algorithm.
    g: forward adjacency list
    src, dst: source and destination indices
    k: number of paths to return
    n: number of vertices
    Returns list of up to k path costs in non-decreasing order.
```

```
"""
# Build reverse graph for backward Dijkstra
revg = [[] for _ in range(n)]
for u in range(n):
    for w, v in g[u]:
        revg[v].append((w, u))
# Run Dijkstra from dst on reverse graph
d, p = dijkstra(revg, dst, n)
if d[src] == INF:
    # No route exists
    return []
# Build shortest-path tree from parents
tree = [[] for _ in range(n)]
for u in range(n):
    if p[u] != -1:
        tree[p[u]].append(u)
# h[u] will point to a heap of sidetrack edges for u
h = [None] * n
queue = [dst]
for u in queue:
    seen_parent = False
    # Explore all outgoing edges of u in forward graph
    for w, v in g[u]:
        if d[v] == INF:
            continue  # unreachable branch
        cost_diff = w + d[v] - d[u]
        # Skip the main tree edge once
        if not seen_parent and v == p[u] and cost_diff == 0:
            seen_parent = True
        else:
            # Insert this sidetrack into u's heap
            h[u] = EHeap.insert(h[u], cost_diff, v)
```

```
        # Propagate heap pointer to children in shortest-path tree
        for v in tree[u]:
            h[v] = h[u]
            queue.append(v)
    # The very shortest path cost
    result = [d[src]]
    # If no sidetracks available at src, only one path exists
    if not h[src]:
        return result
    # Min-heap of (total_cost, heap_node)
    pq = [(d[src] + h[src].key, h[src])]
    # Extract up to k paths
    while pq and len(result) < k:
        total_cost, node = heapq.heappop(pq)
        result.append(total_cost)
        # Follow the sidetrack chain: main branch first
        if h[node.value]:
            heapq.heappush(pq, (total_cost + h[node.value].key, h[node.value]))
        # Then the siblings in the heap
        if node.left:
            heapq.heappush(pq, (total_cost + node.left.key - node.key, node.left))
        if node.right:
            heapq.heappush(pq, (total_cost + node.right.key - node.key, node.right))
    return result
```

The key to Eppstein's efficiency is that it avoids recomputation of shortest paths by reusing the SPT and efficiently merging by using leftist heaps. This design results in a highly efficient method for determining multiple shortest paths in a graph.

Summary of Programming Challenge:

Our programming challenge reframes the k-shortest-paths problem as finding a single path which happens to be the k-th shortest. Given an input graph, start and end nodes, and a

number of people to transport, along with the constraint that no two people may arrive at the same time, the task is to return the path taken by the **last** person. Solving this problem requires computing many shortest paths, usually more than 2 * k, since any paths found that are the same length as the last one must be discarded. Because the runtime of a naive solution scales with k log k, Eppstein's algorithm is almost required for an efficient solution, even more so than for the traditional k-shortest paths problem. Our intention is to demonstrate how much more efficient it is to retrieve a large number of paths from Eppstein's algorithm than by trying to compute them one at a time.

There are two twists that separate our challenge from the traditional application of Eppstein's algorithm: all paths must have distinct lengths, and the algorithm must return the actual path taken instead of the length alone. The first twist is trivial to account for but allows us to more easily leverage the fast scaling of Eppstein's algorithm with large values of k. The second twist adds new and interesting layers to the computation, since the algorithm will now have to keep track of all sidetrack edges taken while it runs, then use them along with the Djikstra's predecessor tree to build the full path. Doing this for all k paths would multiply the runtime by a factor of n, which is why we only ask for a single path.

Key Ideas for Solving Programming Challenge:

Students will need to implement at least a simplified version of Eppstein's algorithm, which is already quite challenging on its own. They may choose to modify it to not even consider multiple paths that share the same length, but can bypass that by simply discarding them as they are encountered. The hardest challenge will be figuring out how to reconstruct the path itself. Students will realize quickly that they are already examining graph edges as they go to compute lengths and can store these edges somehow. Integrating this storage into the priority queue is

tricky but will yield a list of sidetrack edges taken. Students will then have to figure out that these edges do not represent the full path, and do some additional tweaks to be able to traverse the path in linear time.

These changes are interesting because they extend the traditional k-shortest-paths problem into a more constrained and practical scenario, making the problem feel more relevant. At the same time, neither takes a significant amount of extra code once you figure it out, and the more complex change (path reconstruction) is even described in the original paper. The complexity of the model solution is the same as Eppstein's proper, $O(m + n \log n + k)$. Although more than k paths need to be generated for a given problem, each possible path is only considered once, and the runtime for the largest possible k value on a given graph can only be larger by a constant factor. Computing the final path takes O(n) time, but is only done once, so it does not affect the overall complexity.

Conclusion:

The implicit representation of paths through sidetrack edges is the key element of Eppstein's algorithm that allows it to tackle the k shortest paths problem in $O(m + n\log(n) + k)$ time. By re-orienting this problem in terms of a single terminal and deriving explicit paths from sets of sidetrack edges, David Eppstein has created an algorithm that gets the next-best path at every heap pop. This approach is crucial to solving the problem for large graphs, as naive approaches with Dijkstra's algorithm have a much higher asymptotic bound of $O(kn\log(kn))$. Our programming challenge is a modified approach to Eppstein's algorithm, where students need to return *only the* k^{th} shortest path, but no two paths may end at the same time, requiring them to learn how Eppstein's reframes the problem with implicit representation to speed up and reconstruct explicit paths.