K Shortest Paths in Graphs

Kenneth Birk Hansen

Bachelor Thesis IMADA

Supervisor: Rolf Fagerberg

Abstract

Denne rapport handler om problemet med de k-korteste veje i grafteori, som er centralt for både praktiske anvendelser og teoretiske studier. Udgangspunktet er traditionelt at finde den enkelt korteste vej mellem noder i en graf. Dog er der ofte behov for flere næsten optimale løsninger for at opnå øget pålidelighed og fleksibilitet.
Rapporten præsenterer forskellige beregningsstrategier for at løse problemet effektivt, med særligt fokus på David Eppsteins metode. Eppsteins algoritme udnytter avancerede datastrukturer og graftransformationer til at identificere de k-korteste veje med minimal beregningsmæssig overhead.
Desuden introducerer rapporten brugen af quake heaps, en datastruktur der forbedrer køretiden af algoritmen. Resultaterne af denne tilgang anvendes i applikationer som netværksrouting, robotteknik, transportlogistik og videnskabelig databehandling, hvor det er nødvendigt med flere alternative veje for at sikre robusthed og effektivitet.

Introduction

The shortest path problem is a well-known problem that is the core problem in Computer science. The problem focuses on finding the single shortest path between nodes in a graph. However, sometimes this is not the solution that is wanted, For some problems to have multiple near-optimal solutions gives better information about the problem. This is because this can help find a more reliable and flexible path. The k-shortest paths problem expands on this problem by not only seeking the shortest path but also the second shortest, third shortest, and so on, up to the $k'th$ shortest path between a pair of vertices. This expansion allows for a deeper analysis and decision-making process in various applications.

The k-shortest path algorithm in Eppstein's paper uses a combination of advanced data structures and graph transformations to produce an algorithm that can identify the k-shortest paths efficiently. The algorithm restructures the graph into a new form, where paths are represented to make subsequent paths derivable from earlier ones, thereby reducing redundant calculations.

Lastly, we introduce quake heaps which is a data structure that has some improved run times, compared to normal heaps, we use these to improve the run time of the algorithm.

Historical Context and Related Work

Before we can talk about the k-shortest path problem, we need to talk about the shortest path problem. this problem was first solved by Edsger W. Dijkstra in 1956, with his famous Algorithm "Dijkstra's Algorithm" This a fundamental algorithm that is used throughout the world to this day. The k-shortest path problem is then an expansion, this problem can furthermore, be split into two separately different problems. With cycles allowed, and with cycles not allowed. With no cycles allowed Lawler [3] made some improvements on Yen's[4] algorithm. Then with cycles, which is what Eppstein's paper[1] is focusing on.
This paper [2] is working on the k-shortest path problem with cycles allowed and is making a time-complexity improvement by using quake heaps in the Dijkstra's Algorithm, so we go from a time-complexity $m \cdot \log(n)$ to $m + n \cdot \log(n)$.

Applications

Several real-world applications that might benefit from utilizing this algorithm:

Network routing

In telecommunications and data networking, routing decisions can benefit from the opportunity to identify several "good" paths. The algorithm can also find alternative routes in advance of a possible link's congestion or even failure, hence improving the reliability of networking. With the k shortest paths algorithm, it is simpler to balance the load between the different available paths and ensure fault tolerance.

Robotics

In the real world, some types of robots should move from point A to point B quickly. At the same time, they must avoid obstacles when they encounter them. The k shortest paths algorithm for the proposed scheme might assist such types of robotics anywhere in the world. For example, the availability of several possible alternative paths will allow re-routing the path dynamically in real-time to avoid the floating object and its vicinity.

Transportation Logistics

Optimizing routes for delivery vehicles can lead to very large amounts of cost savings and efficiency improvements. Using the k-shortest paths algorithm, logistic planners can generate a set of optimal routes that consider various constraints like traffic, road closures, and delivery time windows, enhancing operational efficiency.

Scientific Computing

In computational biology, the task of sequence alignment can be seen as a path-finding problem in a graph representing the alignment space. The k-shortest paths algorithm can be used to find multiple high-scoring alignments, which is important for understanding biological functions and evolutionary relationships between sequences.

Paper structure

This paper is structured as follows: The **Preliminaries** section introduces concepts and terminology that need to be understood to understand the k-shortest paths problem. **Sidetracks** section explains how the algorithm defines how much a path is longer than another path. The **Heaps and Persistent Data Structures** section, talks about the construction and use of heaps for managing sidetracks. The **D(G) Construction** section describes the creation of a DAG

$D(G)$ and its role in representing paths. The **P(G) Construction** section outlines the transformation from $D(G)$ to the path graph $P(G)$, which is crucial for generating the k-shortest paths. The **Finding the k-Shortest Paths** section details the algorithmic approach to enumerate the k-shortest paths using the constructed path graph $P(G)$. In the **Quake Heaps and Amortized Analysis** section, we introduce quake heaps and provide an amortized analysis of various heap operations.

Preliminaries

Directed Edges

- **Directed Edge**: This is an edge in a graph that has a direction, going from a start vertex (tail) to an end vertex (head). It is represented as an ordered pair of vertices.

Paths and Cycles

- **Path**: A path is a sequence of vertices where each adjacent pair of vertices is connected by an edge.
- **Cycle**: A cycle is a path in which the first and last vertices are the same, and no other vertices are repeated. This means a cycle forms a closed loop. In a directed graph, all edges in a cycle must follow the direction of the edges.

Edge Length and Path Distance

- **Edge Length ($\ell(e)$)**: This is the weight or cost associated with traversing a specific edge in the graph. It is typically a non-negative number.
- **Path Distance**: Path distance is the sum of the lengths of all the edges that make up the path. The shortest path between two vertices is the path with the minimum path distance.

Heaps

- **binary tree**: a tree structure that can at most have two children.
- **min-Heap**: A min-heap is a special type of binary tree: for a min-heap, the value of each node is less than or equal to the values of its children, ensuring that the smallest value is always at the root.
- **Balanced Heaps**: These are heaps that maintain their balance to ensure logarithmic time complexity for insertion and deletion operations. A balanced heap does not need to be a perfectly balanced tree but should be structured to maintain efficiency.

Sidetracks

We define a sidetrack as when you take an edge that is not in the shortest path. First, we have The graph (Figure 1. a) If we want the k path we need to deviate from the shortest path at some point. First, we need the shortest path from each node to our t, our destination vertex. This can be done by reversing every edge in G (G^R) and applying Dijkstra's algorithm to our destination node t. And then reversing every edge again so we have G again. This gives us a single destination shortest path tree. This is seen in (figure 1. b). I want to define lastsidetrack(p) as if there is a path p that uses both edges in T and sidetracks, then if we apply last-sidetrack(p) it removes the last sidetrack from the path.

Now with all the shortest paths in T, we want a way to measure how much longer a path is if we deviate from the shortest path and go into $G - T$. We calculate these sidetracks with the following equation.

$$\delta(e) = \ell(e) + d(\text{head}(e), t) - d(\text{tail}(e), t).$$

- $\ell(e)$: Represents the length of edge e, indicating the immediate cost of traversing this edge.
- $d(\text{head}(e), t)$: Denotes the distance from the head, or endpoint, of edge e to the target vertex t, assuming one continues along the shortest path from this point onward.
- $d(\text{tail}(e), t)$: denotes the distance from the tail, of edge e to the target vertex t, along the shortest path without including edge e.

The function $\delta(e)$ calculates the additional length of the path caused by taking an edge e. It does this by taking the length of edge e and adding the cost to reach t from the head of e.
It then subtracts the cost that would have been incurred if you went from the tail of e directly to t without the detour. These edges are what we call sidetracks. This equation is applied to each edge in $G - T$ that give us Figure 2.

Lemma 1. For any edge e in G, $\delta(e) \geq 0$. For any edge e in T, $\delta(e) = 0$.

Given that e is an edge in the shortest path tree T, the path that includes e from its tail to t is the shortest path. Therefore, we can express the distance from the tail of e to t as the sum of $\ell(e)$ and $d(\text{head}(e), t)$:

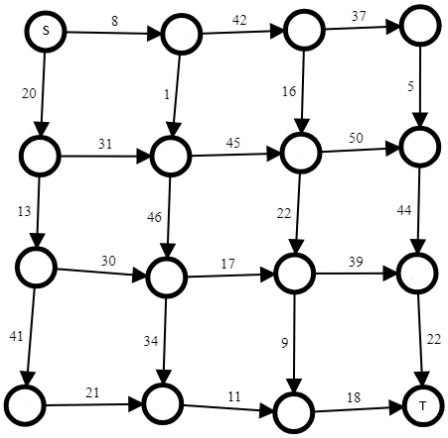
$$d(\text{tail}(e), t) = \ell(e) + d(\text{head}(e), t)$$

Substituting this into the definition of $\delta(e)$, we get:

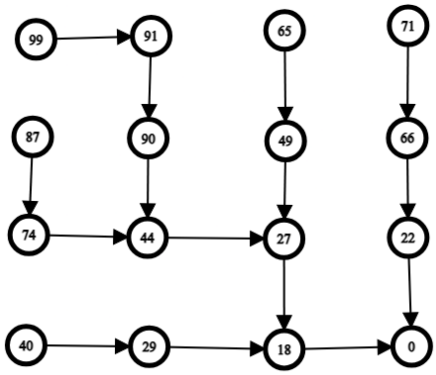
$$\delta(e) = \ell(e) + d(\text{head}(e), t) - d(\text{tail}(e), t)$$

Given our substitution, this simplifies to:

$$\delta(e) = \ell(e) + d(\text{head}(e), t) - (\ell(e) + d(\text{head}(e), t)) = 0$$



(a) Original Graph



(b) single destination shortest path tree, where each number is the "cost" of going from that node to the destination node t

Figure 1: original and single destination shortest path tree

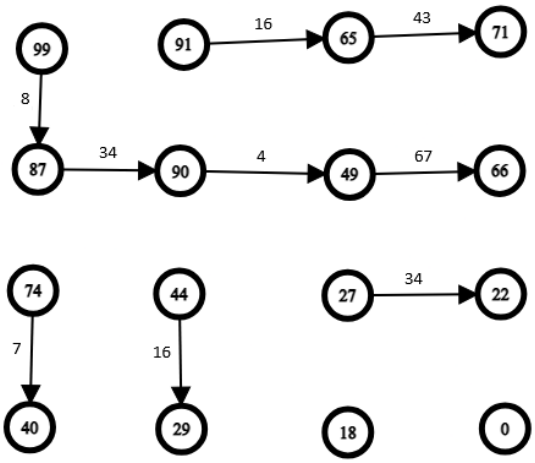


Figure 2: $\delta(e)$ values of edges in $G - T$

This demonstrates why $\delta(e) = 0$ for any edge e within T. The edge e's inclusion in the path contributes precisely to the shortest distance without any additional length, justifying the statement of Lemma 1.

When you are choosing a k shortest path it is uniquely determined by which and how many sidetracks you take. First, it is the shortest path from s to t. In Figure 2, there are 5 different sidetracks that can be chosen from the shortest path. Each of these sidetracks can be seen as a different way down a tree. The root itself is the shortest path with no sidetracks. The roots have 5 children. If you go down one of these you go to a new part of t (figure 2), which has the shortest path. this path also has children which is sidetracks on the shortest path. By doing this repeatedly, a tree is built to describe how you can go through G by which and how many sidetracks you take.

We want a way to calculate the full length of the paths with the shortest paths and the sidetracks. we do this with Lemma 2.

Lemma 2. For any path p from s to t,

$$\ell(p) = d(s,t) + \sum_{e \in \text{sidetracks}(p)} \delta(e) = d(s,t) + \sum_{e \in p} \delta(e).$$

This lemma establishes the relationship between the total length of a path p from s to t, the shortest distance $d(s,t)$ between s and t, and the additional lengths ($\delta(e)$) incurred by taking sidetracks.
Proof for Lemma 2. Consider a path p that diverges from the shortest path by incorporating sidetrack edges. The total length of p, $\ell(p)$, can be broken down into two components: the length of the shortest path from s to t, $d(s,t)$,

and the cumulative additional length introduced by each sidetrack edge in p. Each sidetrack edge e introduces an additional length over the shortest path, quantified by $\delta(e)$. Thus, the total length of path p can be calculated by adding the shortest distance $d(s,t)$ and the sum of $\delta(e)$ for all sidetrack edges in p.

Therefore, we can conclude that:

$$\ell(p) = d(s,t) + \sum_{e \in \text{sidetracks}(p)} \delta(e) = d(s,t) + \sum_{e \in p} \delta(e).$$

This equality holds because $\delta(e) = 0$ for all edges e not in sidetracks, as they are part of the shortest path.

Lemma 3. For any path p from s to t in G, for which sidetracks(p) is nonempty,

$$\ell(p) \geq \ell(\text{prefpath}(p)).$$

Lemma 3 asserts that the length of any path p from s to t that includes at least one sidetrack is greater than or equal to the length of the corresponding prefpath, which is a path that includes all but the last sidetrack of p.

Proof for Lemma 3.

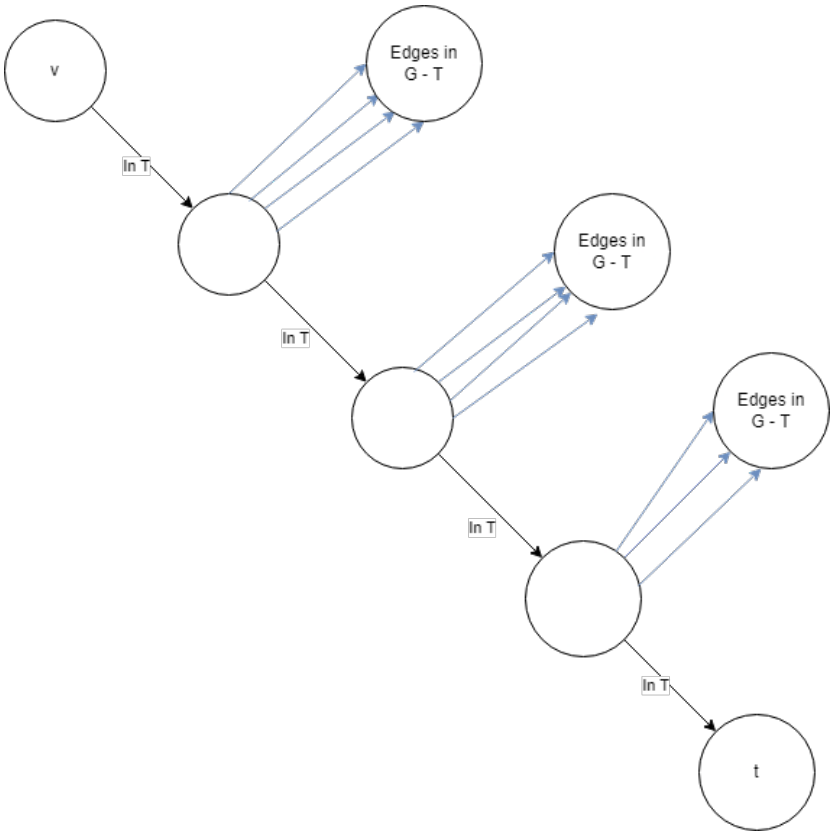
The prefpath of p, denoted as prefpath(p), is essentially p minus its last sidetrack edge. Since $\delta(e)$ represents the additional length over the shortest path introduced by taking edge e as a sidetrack, removing a sidetrack (and thus the associated $\delta(e)$) cannot increase the total path length. Instead, it either reduces the path length or leaves it unchanged.

Therefore, for any path p that deviates from the shortest path by at least one sidetrack, removing a sidetrack to form prefpath(p) results in a path that is at most the length of p, leading to:

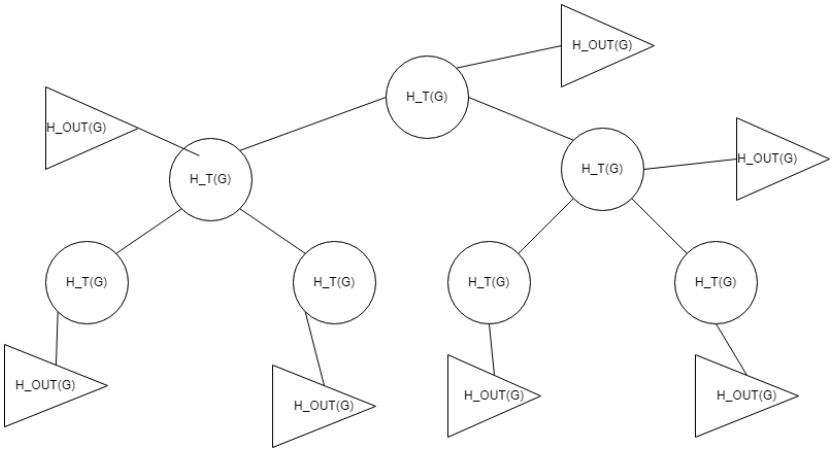
$$\ell(p) \geq \ell(\text{prefpath}(p)).$$

Heap

The path tree discussed, By Lemma 3, is heap-ordered. It is built from the directed graph G, and its nodes represent paths, with the children of a node representing extensions of the path by one additional edge. If the graph G contains cycles, there can be an infinite number of paths due to the possibility of repeatedly traversing the cycles. This means there can be a potentially infinite number of children (paths) for some nodes, making the tree infinitely big. The degree is bounded by the number of vertices, because a vertex cannot go to another vertex twice.



(a) illustration of how each vertex from v to t has sidetracks



(b) H(G)

Figure 3

Building the heaps

First I start on a path from v to t then each of the vertices that are on that path, has a number of sidetracks(this can be seen in Figure 3. a). This is called $out(v)$. Then it builds a 2-min-heap from each of these vertices based on the $\delta(e)$ function. But it is built in a special way, the root only has one child. This heap is called $H_{out}(v)$

Now it only looks at the root of each of these $H_{out}(v)$ min-heaps and we want to build a min-heap out of these roots. Then we want to add the rest of $H_{out}(v)$ onto $H_T(v)$. Here we use the fact $H_{out}(v)$ root only has one child, because it adds one extra child to $H_T(v)$ which is $H_{out}(v)$.
The combination of these two heaps is called $H_G(v)$, it is a normal 2-min-heap, but each node has an extra child that is the entire $H_{out}(v)$ heap. This is illustrated in Figure 3. b.

Persistent

A persistent data structure allows for the preservation and access of previous versions of itself after modifications. In practical terms, when a persistent data structure is modified—such as adding a new node to a tree—the original structure remains intact, and a new version of the structure is created. This new version shares most of its content with the old version, only differing in the parts that have been changed.
This property of preserving past versions without requiring complete copies of the data structure at each modification leads to significant improvements in memory usage and efficiency.
When you are building $H_T(v)$ this needs to be done persistently so it uses the least space. When $H_T(v)$ is forming each new node that is inserted into the tree, a sub-tree is made with only the new node, and all of its parents including the root. All of the remaining nodes that are not in the new tree are referenced, as they are in the previous tree(This can be seen in Figure 4). This saves a lot of space because, for each new iteration, there is a minimum amount of redundant data that is saved.

Building D(G)

Lemma 4. In time $O(m + n \log n)$, we can construct a Directed Acyclic Graph (DAG) $D(G)$ and a mapping from vertices $v \in G$ to $h(v) \in D(G)$, with the following properties:

1. $D(G)$ has $O(m + n \log n)$ vertices.
2. Each vertex in $D(G)$ corresponds to an edge in $G - T$.
3. Each vertex in $D(G)$ has out-degree at most 3.

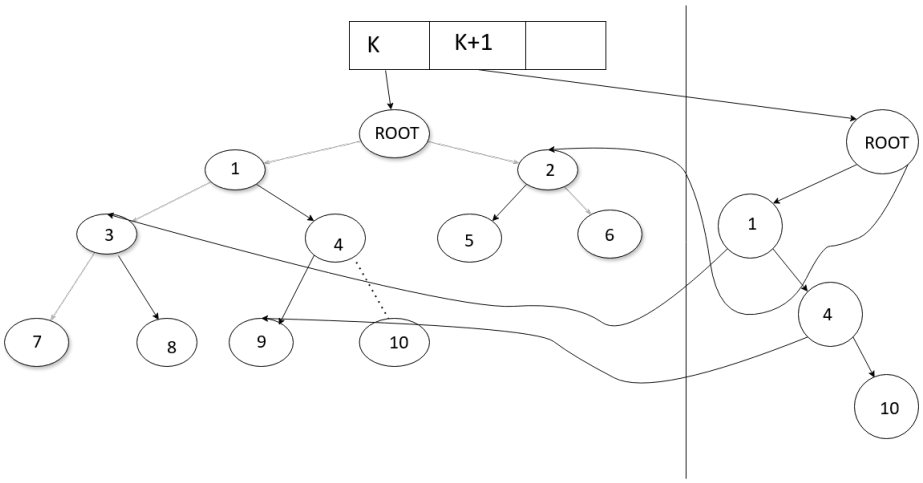


Figure 4: Shows how $H_T(v)$ is built persistently

4. The vertices reachable in $D(G)$ from $h(v)$ form a 3-heap $H_G(v)$ in which the vertices of the heap correspond to edges of $G - T$ with tails on the path in T from v to t, in heap order by the values of $\delta(e)$.

Proof:
(1) The construction of $D(G)$ is built on two sets of heaps: $H_{out}(v)$ and $H_T(v)$. Each node in $H_{out}(v)$ represents edges not in the shortest path tree $G - T$ and contributes one node to $D(G)$. The nodes in $H_T(v)$, which correspond to the shortest path from v to t, contribute $\log_2 i$ nodes to $D(G)$, where i denotes the number of edges in the path from the vertex to t. This logarithmic factor arises due to the binary heap structure of $H_T(v)$, which implies that the height—and therefore the number of nodes contributed—grows logarithmically with the number of edges in the shortest path from v to t.

(2) When incorporating a new edge by inserting $H_{root}(v)$ into the next place in $H_T(v)$, we offset the count by one because $H_{root}(v)$ itself was already included in our $H_{out}(v)$ count. Summing these contributions, the total number of nodes in $D(G)$ is at most $m + n \log_2 n - c$ for some constant c, where m is the number of edges and n is the number of vertices in the graph.

(3) The degree bound of each node in $D(G)$ follows from the construction process, ensuring that every node in $D(G)$ has an out-degree of at most 3.

(4) Mapping each vertex v in G to $h(v)$ in $D(G)$ connects vertices to the root of their corresponding heap $H_G(v)$. The function $\delta(v)$ denotes the extra path length when edge u is used as a sidetrack in G. Each edge (u, v) in $D(G)$ maps back to an edge either in some $H_T(v)$ or in the set of out-edges $H_{out}(v)$. This correspondence ensures that the heap ordering of $D(G)$ naturally arises from

the ordering within these smaller heaps.

Building P(G)

We derive the graph $P(G)$ based on the shortest path tree T rooted at the destination node t. The goal is to represent all paths from the source to the destination efficiently.
In this section, we describe the transformation from the graph $D(G)$ to the path graph $P(G)$. This transformation is crucial for generating the k-shortest paths efficiently.

Constructing the Path Graph $P(G)$

The path graph $P(G)$ is constructed from $D(G)$ as follows:

1. Add a new root node r to represent the source s.
2. Add an edge from r to vertex $h(s)$, which is the root of the heap corresponding to the starting vertex in $D(G)$.
3. For each directed edge (u, v) in $D(G)$, add a corresponding heap edge between u and v in $P(G)$, with a weight equal to $\delta(v) - \delta(u)$. This maintains the ordering of paths according to their costs.
4. For each vertex v representing an edge in $G - T$ with an endpoint at w, add a cross edge from v to $h(w)$ in $P(G)$ with a weight of $\delta(h(w))$.

Properties of the Path Graph $P(G)$

The final graph $P(G)$ has several useful properties:

1. The vertices of $P(G)$ consist of the vertices of $D(G)$ and the new root r.
2. Each vertex in $P(G)$ has an out-degree of at most four.
3. There is a one-to-one correspondence between $s - t$ paths in G and paths starting from the root r in $P(G)$.
4. The length of a path in $P(G)$ corresponds to the length of the original path in G plus the initial shortest path length from the source s.

Lemma 5. There is a one-to-one correspondence between source-destination paths in G and paths starting from the root node r in $P(G)$. This correspondence preserves path lengths.
Proof: Consider an s-t path p in G, this path is defined uniquely by sidetracks(p), the sequence of edges from p in $G - T$. We will show that for any such sequence, there exists a unique path from r in $P(G)$ ending at a node corresponding to lastsidetrack(p). Conversely, any path from r in $P(G)$ corresponds to

sidetracks(p) for some path p

Given a path p in G, construct a corresponding path p' in $P(G)$. If sidetracks(p) is empty (i.e., p is the shortest path), let p' be the single node r. Otherwise, form a path q' in $P(G)$ corresponding to prefpath(p), using induction on the length of sidetracks(p). By induction, q' ends at a node of $P(G)$ corresponding to edge $(u, v) = lastsidetrack(prefpath(p))$. When constructing $P(G)$ from $D(G)$, an edge was added from this node to $h(v)$. Since lastsidetrack(p) has its tail on the path in T from v to t, it corresponds to a unique node in $H_G(v)$, and p' is formed by concatenating q' with the path from $h(v)$ to that node. The edge lengths on this concatenated path correspond to $\delta(lastsidetrack(p))$, and $\ell(p) = \ell(prefpath(p)) + \ell(lastsidetrack(p))$ by Lemma 2. Thus, by induction, $\ell(p) = \ell(q') + \ell(lastsidetrack(p)) = \ell(p')$.

Conversely, to construct an s-t path p in G from a path p' in $P(G)$, create a sequence of edges in G, pathseq(p'). If p' is empty, pathseq(p') is also empty. Otherwise, form pathseq(p') by sequentially taking the edges in G corresponding to the tails of cross edges in p' and adding, at the end, the edge in G corresponding to the final vertex of p'. Since the nodes of $P(G)$ reachable from the head of each cross edge (u, v) are exactly those in $H_G(v)$, each successive edge added to pathseq(p') lies on the path in T from v to t. Therefore, pathseq(p') is of the form sidetracks(p) for some path p in G.

Lemma 6. In $O(m + n \log n)$ time we can construct a graph $P(G)$ with a distinguished vertex r, having the following properties:

- $P(G)$ has $O(m + n \log n)$ vertices.
- Each vertex of $P(G)$ has out-degree at most four.
- Each edge of $P(G)$ has non-negative weight.
- There is a one-to-one correspondence between s-t paths in G and paths starting from r in $P(G)$.
- The correspondence preserves lengths of paths in that length ℓ in $P(G)$ corresponds to length $d(s, t) + \ell$ in G.

Proof: The upper bound of time, size, and out-degree follow from lemma 4. The non-negativity follows from the fact it is heap-ordered, which is proven in Lemma 4. Lastly, the correctness of the correspondence between paths in G and P(G) is shown in Lemma 5.

The graph $P(G)$ is a crucial structure for enumerating the k-shortest paths. The transformation from $D(G)$ to $P(G)$ provides a systematic way to represent paths efficiently while preserving the path ordering by cost.

H(G)

Lemma 7. $H(G)$ is a 4-heap in which there is a one-to-one correspondence between nodes and s-t paths in G, and in which the length of a path in G is d(s; t) plus the weight of the corresponding node in H(G).

$H(G)$ is made from $P(G)$ such that each path from the starting vertex s is represented. The root in $H(G)$ is the empty path, and then its children are all the vertices that are directly connected to s. Each of their children represents the vertices that are connected to them, respectively. $H(G)$ can naturally become large, so we construct it incrementally as we traverse it, building only the parts needed for the next step.

Finding the k Shortest paths

With the construction of $P(G)$ completed, now I show how to utilize this structure to efficiently find and list the k shortest paths from a source vertex s to a target vertex t. The final component of our approach involves applying a breadth-first search (BFS) on $P(G)$ to enumerate these paths in order of their path lengths.

Theorem 1.

Theorem 1: In time $O(m + n \cdot \log(n))$, we can construct a data structure that will output the shortest paths from s to t in a graph in order by weight, taking time $O(\log(k))$ to output the k'th path.

Proof: First I start with traversing through $H(G)$ When retrieving the $k'th$ shortest path, the algorithm explores all paths from the shortest up to the $k'th$ shortest. This exploration is efficiently managed using a priority queue. This queue is a dynamic data structure that contains only the children of the current vertex being evaluated at any given moment during the traversal. As the algorithm goes from one vertex to the next, it removes the current vertex from the priority queue and simultaneously adds the children of this vertex.
This method is efficient due to the structure of the graph $P(G)$, which has a maximum degree of four. This ensures that the data structure does not grow exponentially.
Now we go to path extraction For each path p' identified in $P(G)$, the corresponding path p in the original graph G can be constructed. The link maintained between P(G) and G ensures that the path costs are preserved, and thus, the k'th path extracted from $H(G)$ is the k'th shortest path in G.
And then the Logarithmic Output Time is made once you reach a vertex in $P(G)$ representing the end of an st path, the actual path in G can be detailed in $O(\log(k)$, where k is the index of the path. This speed comes from the fact that each path is represented as a sequence of edges.

Quake heaps

Quake heaps utilize tournament trees, where each element is stored in exactly one leaf, and each internal node's value is the minimum of its children's values. In trees all paths from any node to a leaf have the same length, maintaining balance by requiring all nodes at a given depth to have the same height.

Operations

Insert

Inserting an element is straightforward: Create a new single-node tree for the element.

Decrease-key

To decrease the key of an element:

- Identify the highest node containing that element and cut this subtree, creating a new tree.
- Adjust the element's value, which avoids updating all ancestor nodes.

Delete-min

To perform a delete-min operation:

- Identify the root node with the smallest value (involves comparing all tree roots).
- Remove the path of nodes containing this minimum value.
- Rebalance the heap by linking trees of the same height.

Balancing Strategy

Quake heaps use a lazy balancing strategy, allowing some imbalance and rebuilding parts of the structure when it becomes too imbalanced. They maintain a condition:

$$n_{i+1} \leq \alpha n_i$$

Where α is a constant between $\frac{1}{2}$ and 1.
If violated, it triggers a "seismic" event where the structure above the imbalanced point is removed and later rebuilt.

Amortized Analysis

Amortized analysis is a technique for analyzing the time complexity of algorithms, particularly those involving data structures. Unlike worst-case analysis, which looks at the maximum time an operation can take, or average-case analysis, which considers the expected time over all inputs, amortized analysis provides a more refined perspective. It evaluates the time complexity by considering a sequence of operations and averaging their cost over time. This method is particularly useful for operations that have occasional expensive costs but are generally efficient.

Potential Method

One of the most common techniques in amortized analysis is the potential method. The potential method involves defining a potential function, denoted as Φ, which maps the state of the data structure to a real number. This potential function represents the "stored energy" or "prepaid work" of the data structure. The amortized cost of an operation is then determined by the actual cost of the operation plus the change in the potential function.

Amortized Analysis of Quake Heaps

Potential Method for Quake Heaps

To perform amortized analysis on Quake Heaps, we define an appropriate potential function that captures the state of the heap. The potential function helps us understand the amortized cost of operations by accounting for the "stored energy" in the data structure.

Definition and Application

Let's denote:

- $\Phi(Q)$: Potential function of the Quake Heap Q.
- c_i: Actual cost of the i-th operation.
- $\hat{c_i}$: Amortized cost of the i-th operation.

The amortized cost $\hat{c_i}$ for an operation is given by:

$$\hat{c_i} = c_i + \Delta\Phi$$

where $\Delta\Phi = \Phi(Q_{\text{new}}) - \Phi(Q_{\text{old}})$.

For a sequence of n operations, the total amortized cost is:

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \Delta\Phi) = \sum_{i=1}^{n} c_i + \Phi(Q_{\text{end}}) - \Phi(Q_{\text{start}})$$

If the potential function Φ is chosen such that $\Phi(Q_{\text{end}}) \geq \Phi(Q_{\text{start}})$ and Φ is always non-negative, then the total amortized cost provides an upper bound on the actual cost of the operations.

Amortized Cost of Insertion in Quake Heaps

Consider the insertion operation in a Quake Heap. Typically, the insertion involves placing the new element in a suitable position, creating a new tree of size 1. The number of trees in the collection increases by 1, but can be reduced by linking at a convenient time later.

Define the potential function Φ as the total number of elements in the heap. Each insertion increases the potential by 1, and the actual cost is $O(1)$.

The amortized cost $\hat{c_i}$ of an insertion operation is:

$$\hat{c_i} = c_i + \Delta\Phi = O(1) + 1 = O(1)$$

Amortized Cost of Decrease-Key in Quake Heaps

For the decrease-key operation, the actual cost involves reducing the key of an element and potentially restructuring the heap. Let x be the highest node that stores the element. Instead of updating all the ancestors of x, we perform a cut at x, creating a separate new tree.

The potential function Φ is based on the number of elements and their positions. The change in potential $\Delta\Phi$ due to a decrease-key operation is small, and the actual cost c_i is $O(1)$.

The amortized cost $\hat{c_i}$ of a decrease-key operation is:

$$\hat{c_i} = c_i + \Delta\Phi = O(1) + 1 = O(1)$$

Amortized Cost of Delete-Min in Quake Heaps

The delete-min operation removes the minimum element from the heap. This operation involves removing the path of nodes that store the minimum element and potentially performing linking operations to reduce the number of trees.

Define the potential function Φ as the total number of elements in the heap. Let T be the number of trees, and B be the number of degree-1 nodes. The potential function is defined as $\Phi = N + T + \frac{1}{2\alpha-1}B$, where N is the number of nodes.

The actual cost of delete-min is $O(\log n)$ due to the restructuring. The change in potential $\Delta\Phi$ is the net change in the number of trees and degree-1 nodes. The maximum height, and thus the length of the path, is $O(\log n)$. Linking does not create degree-1 nodes, so the change in B is non-positive.

The amortized cost $\hat{c_i}$ of the delete-min operation is:

$$\hat{c_i} = c_i + \Delta\Phi = O(\log n) + (\text{change in potential})$$

Breakdown of Components

- **Dijkstra's Algorithm with normal heap:** when it is implemented with normal heaps, the three operations mentioned above, have the time complexity $\log(n)$ which means the amortized time of Dijkstra's is $m \cdot \log(n)$
- **Dijkstra's Algorithm with Quake heaps:** When implemented with Quake heaps, the runtime for Dijkstra's algorithm is simplified to $m + n \cdot \log(n)$, where m is the number of edges, and n is the number of vertices. Quake heaps improve efficiency due to their constant time complexity for insert and decrease-key operations, and $\log(n)$ time for delete-min operations.
- **Building the Graph Structure:** The construction of the graph $P(G)$ includes the setup of all necessary data structures for path selecting, amounting to a runtime of $m + n \cdot \log(n)$.
- **Extracting K Shortest Paths:** The complexity for extracting k path is $k \cdot \log(k)$, where k is the number of paths to extract. This accounts for managing paths within a heap or similar structure sorted by their total weights.

Total Amortized Time Calculation

Combining these components, the total amortized runtime for the algorithm using normal heaps and then Quake heaps can be summarized as:

$$\text{Total "Normal" Runtime} = (m \cdot \log(n)) + (m + n \cdot \log(n)) + (k \cdot \log(k))$$

$$\text{Total "Quake" Runtime} = (m + n \cdot \log(n)) + (m + n \cdot \log(n)) + (k \cdot \log(k))$$

This simplifies to:

$$2m + 2n \cdot \log(n) + k \cdot \log(k)$$

The time complexity with the "normal" heap is growing faster because $n \leq m$ so in the best case the run time is equal.

References

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

[2] Kenneth Birk Hansen. K shortest paths in graphs, 2024. Bachelor's Thesis, University of Southern Denmark.

[3] Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[4] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.