IT UNIVERSITY OF COPENHAGEN

Bachelor Thesis

# SWANN

Participating in the Big-ANN-Challenge

Christoffer Jakob Woldbye Romild (chwo@itu.dk)
Thomas Hallundbæk Schauser (hasc@itu.dk)
Joachim Alexander Borup (aljb@itu.dk)

July 16, 2023

**Supervisor**
Prof. Martin Aumüller

**STADS**
BIBAPRO1PE

* Generated by Lexica at https://lexica.art/aperture

**Abstract**

Locality-Sensitive Hashing (LSH) plays a crucial role in approximate nearest neighbour (ANN) search and similarity-based queries. In this paper, we present a study on the theory, implementation, and performance analysis of an LSH algorithm for indexing and searching high-dimensional binary vectors, drawing great inspiration from PUFFINN (Aumüller et al., 2019). We begin by introducing the foundational concepts of LSH, including hash families, bucket distribution, and failure probability. We describe the design and implementation of our LSH-based system, leveraging techniques such as trie-based indexing and efficient bucket distribution. To evaluate the effectiveness of our approach, we conduct benchmarks using the LAION2B dataset, showcasing the accuracy and efficiency of our solution. Our results highlight the importance of careful design choices in hash families. Overall, our study provides insights into the theory, implementation, and performance of our LSH algorithm for efficient indexing and retrieval of binary vectors, contributing to the field of similarity search and approximate nearest neighbour techniques.

The SWANN source code is available at GitHub

# Contents

# 1 Introduction

The problem of finding the approximate nearest neighbours (ANN) in high-dimensional space at a billion-scale (Big ANN) is a fundamental challenge in computer science that arises when you need to find the nearest neighbours of a query point in a large data set (Simhadri et al., 2021). This problem is evident in modern systems such as image search (Yona, 2018), along with recommendation and search engine systems (Bawa et al., 2005). Optimizing the solution regarding memory and performance is essential. The brute force approach of computing the distance between the query point and all other data points is computationally expensive and infeasible for billion-scale data sets as it scales linearly with the input size. Tree-like approaches, such as k-d trees, have been found to scale poorly with a large number of dimensions - dealing with high-dimensional space is also known as the "curse of dimensionality" (Datar et al., 2004). Locality-Sensitive Hashing is a hash based approximation technique originally proposed by Gionis et al. (1999) to overcome this challenge. Since then, numerous other papers have researched this topic. We aim to build on the work of PUFFINN (Aumüller et al., 2019) to research the performance of LSH under the hamming distance metric on 100 Million 1024-dimensional binary vectors. Furthermore, to evaluate our solution, we aim to participate in task C of the SISAP 2023 Indexing Challenge to evaluate the performance of our approach against other state-of-the-art techniques.

## 1.1 The SISAP 2023 Indexing Challenge

The Similarity Search and Applications (SISAP) competition evaluates the performance of state-of-the-art methods in similarity search and related applications under three common tasks A, B, & C.

### 1.1.1 Task C

Task C of the SISAP 2023 Indexing Challenge is about finding the k nearest neighbors of a query point under the hamming distance metric. There are rankings for multiple data-set sizes, but our goal is to provide a solution for the largest dataset containing 100 Million 1024-dimensional binary vectors.

The challenge ranks the solutions based on the best average queries-per-second when run on 10.000 unique query points. The challenge's evaluation is done on the latest Ubuntu version with Docker using a 32-core Intel(R) Xeon(R) CPU E7-4809 workstation with 512GiB of RAM without GPU1.

# 2 Preliminaries

## 2.1 Binary vectors

A binary vector represents a point in Hamming space, which is a conceptual framework where each dimension is represented by a single bit. It encompasses all possible points in a given space, with the number of dimensions denoted as $D$. In a Hamming space with $D$ dimensions, there are $2^{\mathcal{D}}$ distinct points. Consequently, a binary vector can be described as a collection of $D$ bits, each either 0 or 1, and described by $f0\,,1g^{\mathcal{D}}$.

The measure of dissimilarity between two points in Hamming space is known as the Hamming distance. It quantifies the number of differing bits between two points – that is, the sum of their differing bits. In programming terms, it can be determined by performing an XOR operation between two binary vectors and counting the number of set bits.

## 2.2 $(\mathtt{k}, \delta, \mathtt{q})$-NN Problem Definition

Let $\mathtt{P} = f\mathtt{p}_1, \cdots, \mathtt{p}_{\mathtt{N}}g$ define a dataset of $\mathtt{N}$ points in the Hamming space and let each point be a $D$-dimensional binary vector. Furthermore, let $\lambda(\mathtt{a}, \mathtt{b})$ denote the Hamming distance metric between point $\mathtt{a}$ and $\mathtt{b}$. The nearest neighbour of a point $\mathtt{q}$ can then be defined to be the point $\mathtt{p}_{\mathtt{i}}$ $2$ $\mathtt{P}$, such that $\lambda(\mathtt{q}, \mathtt{p}_i) \leq \lambda(\mathtt{q}, \mathtt{p}_n)$ for $\mathtt{n} = f1, \cdots, \mathtt{N}g$.

Given $\mathtt{P}$ and $\lambda(\mathtt{a}, \mathtt{b})$ the $(\mathtt{k}, \delta, \mathtt{q})$-NN problem is about finding $\mathtt{k}$ points in $\mathtt{P}$, such that each is among the $\mathtt{k}$ closest points to $\mathtt{q}$ with a probability of at least $1 - \delta$.

## 2.3 Solution requirement elicitation

To participate successfully in Task C we have elicited the following solution requirements for `SWANN`.

OS The solution must work on the latest Ubuntu Linux.

Dataset The solution must be able to handle a dataset containing up to 100 Million of 1024-dimensional binary vectors each packed into 16 unsigned 64-bit integers, using a `.h5` file encoding.

Queries The queries are for the 10 nearest-neighbours under the hamming distance metric. The queries consist of 10.000 unique 1024-dimensional binary vectors.

Output The indices of the kNN for all queries packed into a single h5 file.

Memory The entire solution must be able to run on 512 GB of memory for 100 Million 1024 dimensional points [1]

Time Must be able to build the index and answer all queries within 12 hours.

Quality The average recall for the queries must be above 90%.

No Network The solution must not use network to call any external API beyond the installation phase.

# 3 Theory

## 3.1 Locality-Sensitive Hashing

Locality-Sensitive Hashing (`LSH`) is a technique used to solve the nearest neighbor search problem in high-dimensional spaces originally introduced by Gionis et al. (1999). In a traditional sense, a perfect hash function would minimise collisions by generating uniformly spread hash values, independent of the input values. The goal of `LSH`, on the other hand, is to hash input into a set of clusters, such that locality-similar points are more likely to collide (i.e. hash to the same value). By doing so, the nearest neighbor of a query point `q` can be efficiently found by examining the points that hash to the same cluster as `q`. `LSH` is particularly useful for very large datasets, where an exhaustive search would be impractical or impossible. The functions that exhibit this locality-sensitive behavior are part of a special family of hash functions that are designed to preserve pairwise similarity - Locality-Sensitive hash families.

## 3.2 Hash families in LSH

When creating the hash family for the `LSH` algorithm, it is important to ensure that the hash functions are independent and uniformly distributed. If they are not, then the produced hash values may not be sufficiently diverse, leading to poor bucket distribution and a poor performance of the `LSH` algorithm. Additionally, it is important to consider the number of hash functions used in the hash family, as these can impact the accuracy and performance of the algorithm.

The hash families used in `LSH` have to satisfy a specific set of properties. These properties are what separates the hash functions in `LSH` families from regular hash functions. Commonly used cryptographic hash functions are designed to minimize hash collisions for closely related sets of input, whereas LSH families are designed to maximize hash collisions for data points close

[1] https://sisap-challenges.github.io/evaluationmethodology/

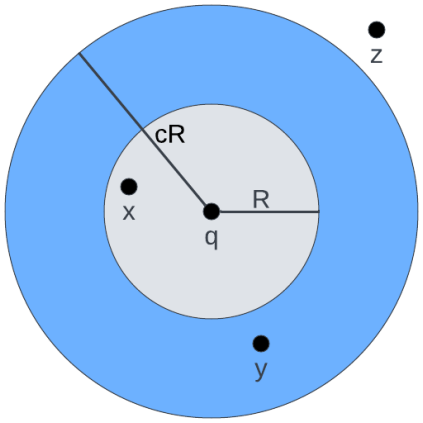


**Figure 1:** *Thresholds and probabilities*

to each other in their respective metric space. Definition 1 explains this formally.

**Definition 1** *A family of hash functions are locality-sensitive for points* $p$ *and* $q$ *in a metric space if and only if the following conditions hold.*

$$\lambda(p,q) \leq R \;!\; Pr[hash(p) = hash(q)] \geq P_1 \tag{1}$$

$$\lambda(p,q) \geq cR \;!\; Pr[hash(p) = hash(q)] \leq P_2 \tag{2}$$

*Where* $R$ *is a threshold,* $c$ *is an approximation factor, and* $Pr[E]$ *is the probability of an event* $E$ *occurring.*

Figure 1 offers a visualization of these probabilities for $f\,x, y, z\,g \in P$ compared to $q$. Given $\lambda(q,x) \leq R$ the probability that $q$ and $x$ will have colliding hashes is at least $P_1$. By this, all points located within the gray inner circle in Figure 1 will have probability of at least $P_1$ of being hashed into the same bucket as $q$.
The probability that $z$ and $q$ will collide is at most $P_2$, for all points outside the blue circle, as it holds that $\lambda(q,z) \geq cR$.
Points at a distance between $R$ and $cR$, generalized to all points in the blue ring, such as the distance between $y$ and $q$, have an unknown probability of collision. The locality-sensitive hash families $H$ all adhere to the property that $P_1 > P_2$ (Jure Leskovec, 2014).

### 3.2.1 Single-bit Hash Family

For binary vector points in *Hamming space*, a common and simple LSH function is to check a random bit[2]. Given a binary vector $p = f0, 1g^{\mathcal{D}}$, let

[2]This is also the LSH family we have used for SWANN.

the notation p[i] denote the value of the i'th bit of p for i 2 f1, . . . , Dg.

$$\text{single\_bit\_hash}(\mathtt{x}) = \mathtt{x}[i] \tag{3}$$

Equation 3 depicts the single-bit hash function. The idea is to choose i at random, and then hash any point by returning the value of the i'th bit.

As the hamming distance metric directly relates to the value of the bits, it makes sense that the single-bit hash family would be locality-sensitive. In theory, such a hash function would separate all points into two clusters - one where the i'th bit is set, and another where it isn't. Probability $\mathtt{P}_1$ of a single-bit hash, that two points a, b 2 P are hashed to same value, can be calculated by equation 4.

$$\mathtt{P}_1 = 1 - \frac{\lambda(\mathtt{a}, \mathtt{b})}{D} \tag{4}$$

Intuitively, as this hash function only checks a single bit, if $\lambda(\mathtt{a}, \mathtt{b}) = \frac{\mathcal{D}}{2}$ ! $\Pr[\mathtt{a}[j] = \mathtt{b}[j]] = 50\%$, meaning $\mathtt{P}_1$ would have a probability of 0.5 as any given bit j 2 f1, . . . , Dg would have a 50% chance of being set in both a, b. Similarly, if $\lambda(\mathtt{a}, \mathtt{b}) = 0$ ! $\Pr[\mathtt{a}[j] = \mathtt{b}[j]] = 0\%$, all the bits would be shared, hence $\mathtt{P}_1$ would be 1.

By itself, a single-bit hash does not impose a major optimization in clustering points together. However, multiple locality-sensitive hash functions can be combined to create a new LSH function with improved properties. Thereby, it is possible to increase the number of generated clusters. This strategy is used to increase the collision probability of a hash function, while maintaining a relatively low computational cost. Definition 2 formally describes this.

**Definition 2** *For an LSH family F, let G be a new family where each element of G is created by a random subset of d elements from F. Given two points, p q, and a hash g 2 G, g(p) = g(q) iff. h(p) = h(q), 8h 2 g, then the* $P_1$ *probability of two points a, b 2 P can be calculated as:*

$$P_1 = (1 - \frac{\lambda(a, b)}{D})^d \tag{5}$$

Chaining of hashes of an LSH family results in a new LSH family, as the resulting family still satisfies the properties of equations 1 & 2. This concept is further explained and visualized in subsubsection 3.3.4. Our solution uses the single-bit hash family with chained functions.

In the chained LSH family, two points a, b 2 P with $\lambda(\mathtt{a}, \mathtt{b}) = \frac{\mathcal{D}}{2}$ and d = 2

would now have a $P_1$ probability of $0.5^2 = 0.25$. If a and b were closer to each other, such that $\lambda(a, b) = \frac{\mathcal{D}}{10}$ with $d = 2$, the probability would be $P_1 = 0.9^2 = 0.81$.

This strategy naturally decreases $P_2$, the probability of far away points being hashed to the same cluster, as powering a probability decreases it. By utilizing this chaining, the total number of clusters is increased to $2^n$ clusters.

## 3.3 LSH Tries

### 3.3.1 Overview

An LSH-trie is a tree-like data structure originally introduced by Bawa et al. (2005) as a solution to the $(k, \delta, q)$-NN problem.

The idea is to draw on definition 2 to group closely related points into buckets by applying chains of locality-sensitive hash functions. To find out which bucket a point belongs to, an LSH-trie of depth d applies d hash functions $2\ H$, where $H$ is any locality-sensitive hash family. The problem of finding the ANN for a query point q, can then be answered by mapping q to a bucket, and examining all points in buckets nearby.

### 3.3.2 Building the LSH-Trie

To construct the trie, each of the d locality-sensitive hash functions is assigned to a depth $i\ 2\ f1, \cdots, dg$, such that $hash_i$ denotes the hash function assigned to depth i. Any node at depth i will contain child nodes for all possible results of applying $hash_i$ to a point.

The trie term in LSH-Trie derives from the fact that each point $p_n\ 2\ P$ is mapped to a binary string[3] $S_n$ of length d, such that the i'th letter is the result of $hash_i(p_n)$. The mapping should always be surjective such that $jS_nj < D$. To find out which bucket $p_n$ belongs to, the letters of $S_n$ are iterated through from start to end like in an ordinary trie. Figure 2 illustrates the general idea mapping $S_n = 010$ to the bucket at index 2. A node at depth i would visit the right child if $S_n[i] = 1$ and left otherwise. Thereby any path from root to leaf can be thought of as the result of applying the $d -$ hash functions.

If the trie uses binary hash functions[4], the nodes of the trie will conveniently be perfectly balanced, such that each node contains two children, resulting

[3] i.e. a string where each letter $c \in \{0, 1\}$

[4] Hash functions that evaluate to either true or false

in exactly $2^d$ leaf-nodes i.e. buckets. It's worth noting that implementations such as PUFFINN (Aumüller et al., 2019), incorporate hash functions that are non-boolean and as such the resulting LSH-trie will be unbalanced.

### 3.3.3 Querying an LSH-Trie for candidate points

To find the nearest-neighbours to a query point q we first map q to a bucket $S_q$. If the bucket at $S_q$ contains an acceptable amount of candidate points the points in the bucket are returned, and otherwise near matching buckets are examined. An LSH-trie offers a simple way to move to near matching buckets. To move to another bucket which shares the result of exactly $d - h$ hash-functions, we can simply flip h bits in $S_q$. The fewer bits we flip, the more likely the resulting bucket's points are to be near to q. The process of flipping h bits in $S_q$ is repeated, incrementally increasing the number of bits flipped from 1 *!* d, until an acceptable amount of candidate points are extracted.

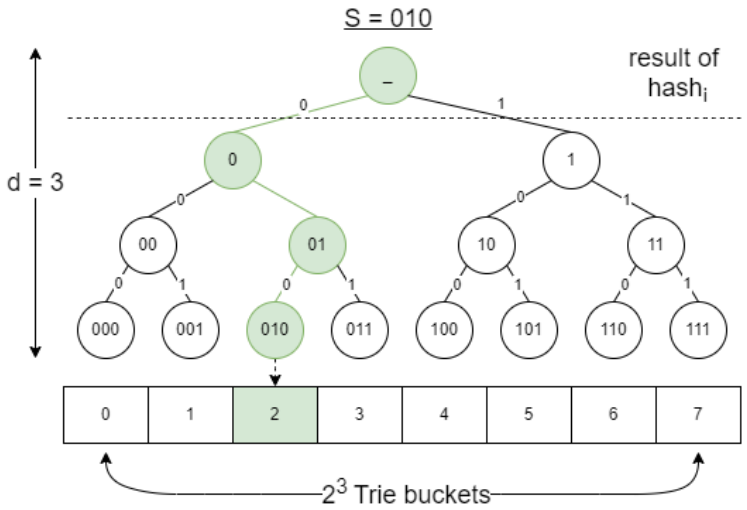


**Figure 2:** *An LSH-trie of height 3. The highlighted path is the result of indexing a point that resulted in the binary string $S = 010$*

### 3.3.4 Failure probability

The failure probability represents the probability that two points that are close to each other will be placed in different buckets and depends on the similarity of the points, as well as the properties of $H$ defined by equations 1 and 2.

Due to the nature of a locality-sensitive hash-family, the deeper the trie, the

higher the probability that closely related points maps to the same bucket, compared to further away points (see 3.2). Thereby, increasing the depth of the trie will decrease the failure probability.

This is the case as probability $P_1$ expresses the minimum average percentage of true positives in terms of nearest neighbours in each bucket. $P_2$ defines the maximum average probability of false positive hash collisions between two points. Since LSH hash families have the property that $P_1 > P_2$ (see 1), $P_1$ would converge towards 0 slower than $P_2$ as $d \; ! \; 1$. However, as a side effect of increasing d, the probability of true nearest neighbours with $\lambda > cR$ colliding decreases, hence resulting in a larger number of buckets with potential candidate points.

## 3.4 LSH Forest

### 3.4.1 Overview

An LSH-forest is a data structure originally introduced by Bawa et al. (2005) to answer k-NN queries. The *forest* part of the name derives from the fact that a collection of L LSH-tries (trees) are maintained as shown in Figure 3 for $L = 2$ and $d = 3$.

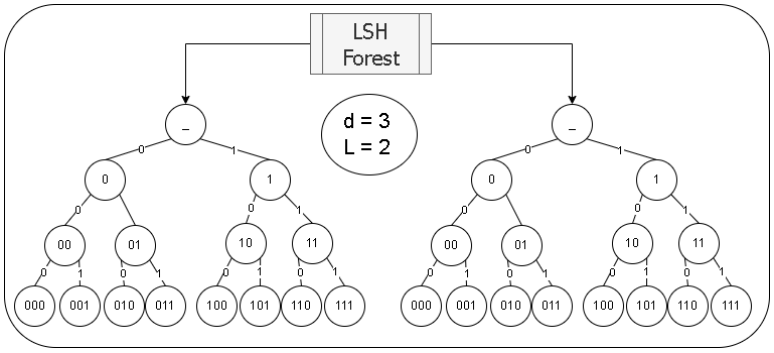


**Figure 3:** *A LSH-Forest consisting of two LSH-tries with* $d = 3$.

### 3.4.2 Building the forest

To build the LSH-Forest data structure the N points are simply inserted into the L-tries as explained in section 3.3.2. It's important that the L LSH-tries all have the same depth d, and have been constructed by choosing d random hash functions from the same locality-sensitive hash family H for each of the tries.

**Choosing d and L** The performance of the forest depends on the choice of d and L. The correct choice of d and L depends on the probabilities $P_1$

| Function | Description |
|---|---|
| CONSTRUCTOR(P, q, k) | Initialize $F_q$ from the input points P, query point q and k. |
| void INSERT(int i) | Insert $p_i$ into $F_q$ |
| int[] EXTRACT_KNN() | Returns a list containing the indices of the k points in $F_q$ with minimal distances to q. |
| int GET_KTH_DIST() | Returns the distance of the kth closest neighbour to q in $F_q$ i.e. $\lambda(q, f_k)$. |

**Table 1:** FOUNDMAP*(P, q, k)*

and $P_2$ for the underlying hash family used. For a single-bit hash function, the depth can be calculated as (Aumüller et al., 2019):

$$d = \left\lceil \frac{\log N}{\log \frac{1}{P_2}} \right\rceil \tag{6}$$

The number of tries in the forest, L, is calculated by:

$$L = \left\lceil P_1^{-d} \right\rceil \tag{7}$$

### 3.4.3 Querying the forest for $(k, \delta, q)$-NN

Querying a point q is handled by extracting candidate points from all of the L-LSH-tries. Upon termination the k points that is nearest to q will be returned.

**Tracking found points** For each query, a set of the k nearest-neighbours found so far is maintained through an auxiliary data structure FOUNDMAP. Let $F_q$ be a FOUNDMAP for query q. Furthermore, let $f_i$ *2* $F_q$ denote the i'th nearest neighbour inserted into $F_q$ such that $\lambda(q, f_i) \leq \lambda(q, f_{i+1})$. Table 1 illustrates the exact specifications of a FOUNDMAP. The running time of queries depend on the underlying implementation of $\text{FOUNDMAP}_q$. See 3.5 for a more thorough explanation of $\text{FOUNDMAP}_q$.

**Stopping queries** Aumüller et al. (2019) defines an exit criteria for the query mechanism, that we define by algorithm 1. The exit criteria stops the query, once the probability of failure (i.e. $P[F]$) is below the given recall $\delta$ (see Algorithm 1).
The exact formula $P_1$ and thereby $P[F]$ depends on the chosen H but equations 8 and 9 displays an example for the single-bit hash family presented previously in subsubsection 3.2.1.

$$P_1 = (1 - \frac{\lambda(q, f_k)}{D})^d \tag{8}$$

$$\mathtt{P}[F] = (1 - \mathtt{P}_1)^{\mathtt{L}} \tag{9}$$

**Algorithm 1** STOP_QUERY$(\lambda(\mathtt{q}, \mathtt{f_k}), \delta, \mathtt{d})$ for the `Single-Bit hash family`

```
if (1 − δ > P[F]) then
    return true
else
    return false
end if
```

**Query pseodo-code** Algorithm 2 denotes the combined query algorithm. Line 1 calls the constructor of an FoundMap for the given query. The while-loop declared on line 3 runs until STOP_QUERY is true, incrementing `hdist` by one for each run (see line 12).
Line 5 sets `hash` to the result of applying the hashes of `T[l]` on `q` (i.e. $\mathtt{S_q}$). The `i`'th iteration queries for candidate points in all buckets within a hamming distance of `hdist` for each of the `L` tries (see line 6). The indices of the points in each of the examined buckets are then inserted into FoundMap on line 8. The number of examined buckets scales with Pascal's Triangle (see Corollary 3), hence the actual implementation applies some additional heuristics to bound the worst-case growth (see section 5).

**Algorithm 2** $(\mathtt{k}, \delta, \mathtt{q})$-NN query with `LSH`-Forest built from `P`

**Require:** $(\mathtt{k} > 0)$, $(0 < \delta \leq 1)$ and `L` LSH-Trie as `T`[1..L]

```
1:  F_q    new FoundMap(P,q,k)
2:  hdist    0
3:  while    STOP_QUERY(F_q.GET_KTH_DIST(), δ, d − hdist) do
4:      for l    1..L do
5:          hash    T[l].HASH(q)
6:          for all bucket 2 T[l].GetBuckets(hash, hdist) do
7:              for all pidx 2 bucket do
8:                  F_q.INSERT(pidx)
9:              end for
10:         end for
11:     end for
12:     ++hdist
13: end while
14: return F_q.EXTRACT_KNN()
```

### 3.5 FoundMap

The primary goals of FoundMap is two fold.

1. Ensure that i insertions of a point $p_n$ *2* P only computes $\lambda(q, p_n)$ once.
2. Bound running times of functions shown in Table 1 by some order of k as the SISAP challenge only queries for the 10-nearest neighbours.

The following sections will show how to achieve these two goals and implement Insert in amortized $O(\log(k))$[5], Knn_Extraction in $O(\log(k) \times k)$ and Get_Kth_Dist in $O(1)$.

Any FoundMap holds a reference to P, q and k, and two auxiliary data structures seen & Knn_Queue.

1. seen[6] is a hashmap containing the indices of found points, allowing amortized constant-time extraction and insertion.
2. Knn_Queue[7] holding a pair containing the indices of the k nearest neighbours to q inserted into FoundMap, and their distance to q. They are ordered by their $\lambda(q, p_n)$ in descending order, such that the top of the queue always points to the k'th nearest neighbour.

Algorithms 3, 4 & 5 shows the implementations.

**Algorithm 3** FoundMap.Insert(int i) in $O(\log(k) + D)$

**Require:** $(1 \leq i \leq N)$

```
1: if   seen.contains(i) then
2:     seen.insert(i)
3:     if Knn_Queue.Size() < k then
4:         Knn_Queue.Push(fλ(q, p_i), ig)
5:     else if λ(q, p_i) < Knn_Queue.Peek().dist then
6:         Knn_Queue.Pop()
7:         Knn_Queue.Push(fλ(q, p_i), ig)
8:     end if
9: end if
```

To adhere to goal 1, line 1 checks whether i *2* Knn_Queue before computing $\lambda(q, p_i)$. As the top of the queue always points to the k'th nearest neighbour, the *if*-statement on line 5 will only be true if a closer k'th nearest neighbour has been found.

[5] assuming the distance computation $\lambda$ evaluates in O(1)

[6] std::unordered_set<int> in C++

[7] std::priority_queue<std::pair<int,int>, std::vector<std::pair<int,int>>, std::less<std::pair<int,int>> in C++

PUSH and POP on a priority-queue runs in logarithmic time to the size of the queue. Hence as $|$KNN_QUEUE$| \leq$ k, the asymptotic running time of insertion is $O(\log(\texttt{k}))$.

**Algorithm 4** FOUNDMAP.EXTRACT_KNN() in $O(\log(\texttt{k}) \times \texttt{k})$

```
1: knn ← new Array[KNN_QUEUE.SIZE()]
2: for i ← {KNN_QUEUE.SIZE(), ··· , 1} do
3:     knn[i] ← KNN_QUEUE.PEEK().INDEX
4:     KNN_QUEUE.POP()
5: end for
6: return knn
```

Algorithm 4 assumes arrays are 1-indexed. The general idea is to transform the KNN_QUEUE into an array, such that the front of the array becomes the nearest neighbour found. Since there can be up to k-POP operations, the overall running time becomes $O(\log(\texttt{k}) \times \texttt{k})$.

**Algorithm 5** FOUNDMAPGET_KTH_DIST() in $\Theta(1)$

```
1: if KNN_QUEUE.EMPTY() then
2:     return INT_MAX
3: else
4:     return KNN_QUEUE.PEEK().DIST
5: end if
```

In case the queue is empty, the GET_KTH_DIST() returns INT_MAX, else it simply returns the distance of the top element in the queue (i.e. the k'th neighbours distance). It is trivial to see that the amortized running time of algorithm 5 is constant, as all called functions are constant and there are no loops.

# 4 Complexity analysis

## 4.1 Building the LSH forest

### 4.1.1 LSH-Trie

The running time of build depends on the time it takes to insert a point into an LSH-Trie, which in turn depends on corollary 1 and 2.

**Corollary 1 (LSH-Trie Bucket indexes)** *Assume* $\texttt{hash}_i \in H$ *for* $i \in \{1, \cdots, \texttt{d}\}$ *each evaluates in* $O(1)$*. Then it takes a LSH-Trie of depth* d $O(\texttt{d})$ *to hash any* $\texttt{p}_n \in P$ *to its bucket index* $\texttt{S}_n$*.*

Corollary 1 follows directly from the fact that an LSH-Trie constructs $\texttt{S}_\texttt{n}$ by applying d hash functions from a locality-sensitive hash family $H$. As each

of these hash functions has constant-time evaluation, the combined time necessary to construct $S_n$ must be $O(d)$.

**Corollary 2 (LSH-Trie-Node indexing)** *A downwards traversal from any node in a perfectly balanced LSH-Trie to one of its children can be done in constant-time $O(1)$.*

Corollary 2 is a fair assumption since the nodes of a perfectly balanced LSH-Trie always have two children per node.

**Theorem 1 (LSH-Trie insertion)** *Given corollaries 1 and 2, then inserting any $p_n \in P$ into an LSH-trie of depth $d$, always takes time linearly proportional to $d$, i.e. $\Theta(d)$.*

Theorem 1 follows directly from $|S_n| = d$, as inserting a $p_n$ into a LSH-trie is done by iterating through the letters of $S_n$ (see 3.3.2).

### 4.1.2 LSH-Forest

**Theorem 2 (LSH-Forest built on P)** *Given a locality-sensitive hash family $H$, building a LSH-Forest of $L$ LSH-Tries with depth $d$ on $P$ takes $\Theta(L \times d \times N)$.*

The building process of an LSH-Forest on P simply involves choosing a random subset of $H$, and inserting the N points L times. Therefore, Theorem 2 follows directly from Theorem 1.

**Theorem 3 (Memory of LSH-Forest built on P)** *An LSH-Forest of $L$ perfectly balanced LSH-Tries of depth $d$ built on $P$ uses at most $O(L \times 2^d + N \times D)$ memory.*

Theorem 3 follows from the fact that there are L perfectly balanced LSH-Tries, each with exactly $2^d$ leaf nodes, and therefore $2^{d+1} - 1$ total nodes (see 3.3.2). The $N \times D$ part of the bound represents the memory footprint of the N points.

### 4.2 $(\mathtt{k}, \delta, \mathtt{q})$-NN queries

**Corollary 3** *Let $S_i$ denote the index of some bucket in an LSH-Trie of depth $d$. There exist up to $\binom{d}{\lfloor d/2 \rfloor}$ buckets with a hamming distance of $0 \leq hdist \leq d$ to $S_i$.*

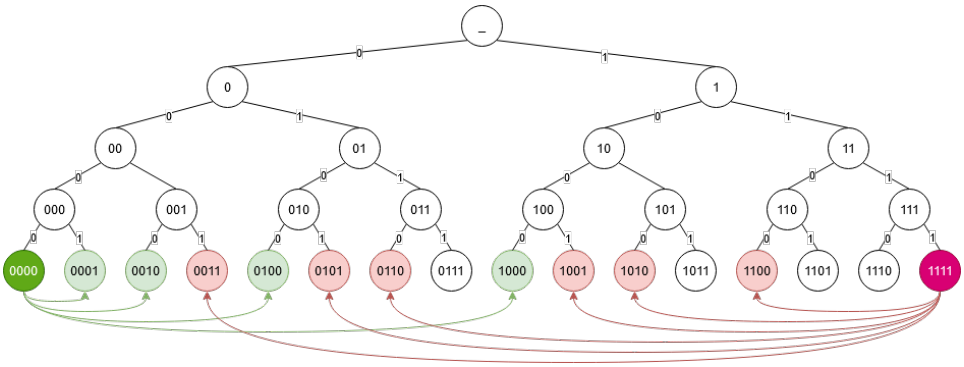


**Figure 4:** *An LSH-trie, with nodes of distance 1 and 2 from nodes 0000 and 1111 highlighted, respectively*

Corollary 3 follows from a correlation between Pascal's Triangle, and the numbers of neighbours within a distance of a given node. Pascal's Triangle is a triangular array of numbers in which each number is the sum of the two numbers directly above it, forming a pattern with various mathematical properties and applications (see Figure 5). The number of neighbours, that are at distance `h` from a node in an `LSH`-Trie of depth `d`, correlates to the `h`'th number in the `d`'th (zero-indexed) row of Pascal's Triangle. For instance, see the `LSH`-Trie of depth 4 in Figure 4, where the neighbours of distance 1 to node 0000 (marked green), and of distance 2 to node 1111 (marked red). The values of the numbers at indicies 1 and 2 in the 4'th row of the triangle in Figure 5 are 4 and 6 (highlighted green and red), meaning there are 4 and 6 neighbours, respectively. It is evident the the middle values of the rows (also highlighted in Figure 4) determines the highest possible number of neighbours, that a node can have at a specific distance. Sperner's Theorem (Reece, 2019) can be used to compute those middle values, stating that $\binom{d}{\lfloor d/2 \rfloor}$ gives the middle value of the `d`'th row of the Triangle. Hence, the worst case is $O(\binom{d}{\lfloor d/2 \rfloor})$.

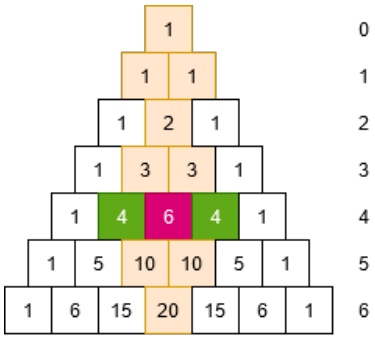


**Figure 5:** *Pascal's Triangle*

**Theorem 4 (Time of GetBuckets)** *It takes at worst* $O(d \times \binom{d}{\lfloor d/2 \rfloor})$ *to retrieve all buckets within any given hamming distance* $0 \leq hdist \leq d$ *in a LSH-Trie of depth* $d$.

Theorem 4 follows directly from the bucket limit imposed by corollary 3 and the time to apply the d hash functions imposed by 1).

**The running time of Algorithm 2** To define the exact running time, we must first determine the number of iterations of the while loop on line 3, which we define as some value Q in the worst-case. The value of Q depends entirely on STOP_QUERY$(\lambda(q, f_k), \delta, d)$, and therefore Equations 8 & 9. Setting a tight bound on the value of Q is outside the scope of this project, but it must be true that $Q \leq d$, since $Q = d$ would constitute checking every single bucket in all of the tries.

Theorem 5 offers a loose bound on the running time.

**Theorem 5 (Time of** $(k, \delta, q)$**-NN with LSH-Forest)** *A* $(k, \delta, q)$*-NN query in an LSH-Forest built from* $P$ *takes at most* $O(d \times \binom{d}{\lfloor d/2 \rfloor} \times L \times \log(k) \times B \times Q)$*, where* $B$ *denotes the maximum number of points in any bucket.*

For each iteration of the while loop, algorithm 2 at worst extracts $L \times \binom{d}{\lfloor d/2 \rfloor}$ buckets in $O(L \times \binom{d}{\lfloor d/2 \rfloor} \times d)$ (see theorem 4). For each of those it inserts up to B points into $F_q$, and it follows from algorithm 3 that inserting a single point into $F_q$ takes $\log(k)$. Therefore, the worst-case running time of queries must be $O(d \times \binom{d}{\lfloor d/2 \rfloor} \times L \times \log(k) \times B \times Q)$.

At first glance, the memory and time requirement of the solution seems astronomical (see Theorem 3 & 5). The following implementation section offers search heuristics, which aim to reduce the running time. Furthermore, the memory footprint is reduced by applying spatial optimizations.

# 5 Implementation

## 5.1 Overview

The source code for the solution has been written in C++23 without any external libraries such as Boost[8]. All parts of the code runs synchronously on a single-core.

The overall system architecture can be divided into two processes: building and querying. Figure 6 offers an overview of our building process. Our

[8] Aside from H5, GoogleTest, and GoogleBenchmark for loading data, unit testing and benchmarking, respectively.

building process aims to reduce B[9] by applying a rebuilding optimization for the LSH-tries explained in further details in 5.3.2.

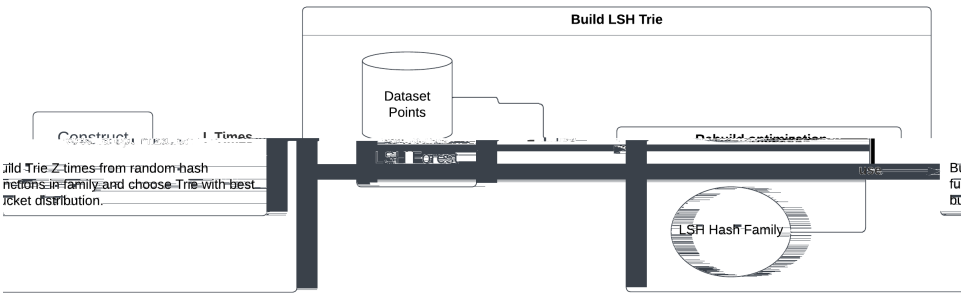


**Figure 6:** *SWANN BUILD: Overview of the building process*

## 5.2 Binary vector

We have implemented points by a class POINT<$D$>that extends std::bitset from C++'s standard library. This implementation offers a portable and easily testable solution, that still allows us to manipulate points by bit operations.

## 5.3 LSH-Trie

For our implementation of an LSH-trie we hold the point buckets as linked-list of point indices. Conveniently, the hash values of the leaf nodes of a trie of depth d correspond to the binary representations of the integers between 0 and $2^d - 1$, as is evident in Figure 2. Thus, a simple array of buckets of length $2^d$ can be used to represent all buckets in a trie of depth d. This allows for very efficient retrieval, since arrays can be indexed in constant time. For instance, if you were to retrieve the adjacent buckets of a data point whose hash value is 010, you simply retrieve the buckets from the array at indices 110, 000, and 011.
To efficiently iterate through all buckets which share $d - h$ hash-functions, we hold a pre-computed array of bit-masks indexed by h. The bit-masks for h is an ordered-set of all possible permutations containing h set bits for a length d binary string. Table 2 illustrates the masks for $d = 3$. It follows directly from corollary 3 that the maximum number of masks for any depth $d$ is $\binom{d}{\lfloor d/2 \rfloor}$.

[9] i.e. The maximum number of points in any bucket

| Depth | Masks |
|---|---|
| 0 | “000” |
| 1 | “001”, “010”, “100” |
| 2 | “011”, “101”, “110” |
| 3 | “111” |

**Table 2:** *Bit-Masks for a LSH-Trie with* $d = 3$

### 5.3.1 Array vs. HashMap

Initially we kept an array of vectors for representing the buckets of a trie, where the hash of a point was the index of the respective bucket.
As an example, adding a point to the trie, would look something like: TRIE[HASH(POINT)].APPEND(POINT). Once we tested our solution on larger datasets, which required a greater trie depth, the memory consumption of this naive implementation became a problem.

As the trie can be visualized as a binary tree, where we would only store the leaf nodes, each consecutive depth would generate two times the previous number of buckets. For example, a trie with depth 15 would generate $2^{15}$ buckets. At this depth, only 33.000 buckets are created per trie, but for the large dataset of the SISAP challenge, we strived for a trie depth of approximate 30. This depth would require each trie to contain over $10^9$ buckets - even if each point was hashed to a unique bucket, more than 90% of the buckets would remain empty.

The C++ memory consumption for an empty bucket is the size of the empty vector, plus the size of the pointer to the vector $24 + 8 = 32$bytes

Thereby for our naive implementation, a trie with depth of 30 would consume around $10^9 \times 32$ bytes ' 32 GB of memory for empty buckets alone. As such, building an LSH-forest would be infeasible.

Similarly to how one would optimize the memory of an ordinary trie, we solved this problem by simply replacing the array implementation with hash maps[10]. This was a straight forward replacement, as the implementation of the data structure is hidden behind a layer of abstraction.

Instead of using the hash of a point as an index in an array, the hash is now used to map to a bucket. Though this is a separate data structure, we still visualize each hash as an index. The only difference being, that the bucket would only be instantiated once a point is hashed to said bucket.

[10] For the hash map we used the C++ standard library container unordered_map.

Performance-wise no difference was found on the depths we were able to test on using both implementations. The `unordered_map` has amortized constant lookup time, hence using the hash map should not have drastic performance implications as corollary 2 still holds.

The hash map significantly improved the performance of the benchmark using deeper tries, however we were unable to compare the performance of the hash-map and array implementations for these depths, due to said memory issues. The bucket distribution of the different approaches can be seen in figure 10 and figure 11. This optimization allowed us to run benchmarks on depths of 30 without any memory implications.

### 5.3.2 Trie rebuilding optimization

The underlying theoretical motivation for this optimization has been that theorem 5 defined the worst-case running time of each query to grow by a factor of `B`, the number of points in the largest bucket for a `LSH`-Trie.

Let $1 <$ `w` be some integer constant. The `LSH`-Trie rebuilding optimization can be thought of as choosing the trie with the smallest value of `B` among `w` `LSH`-Tries, each built from the same dataset and assigned a unique subset of a locality-sensitive $H$.
How to find the correct value of `w` is discussed further in 7.4.1, where we show results for different choices of `w`.

**Impact of optimization** The trie rebuilding optimization had a large impact on the performance of the solution in both theory and practise. In practice, prior to the optimization we observed significant differences between the performance of each benchmark run. In contrast, after implementing the trie rebuilding optimization the differences decreased, and both the quality and performance of each query improved.
In theory, the rebuilding optimization has slowed down the upper bound for building the `LSH`-forest from theorem 2 to $O(\texttt{w} \times \texttt{L} \times \texttt{d} \times \texttt{N})$.

## 5.4 LSH-Forest

As finding the optimal values for $\texttt{P}_1$ and $\texttt{P}_2$ programmatically for a dataset was outside the scope of this project, we tuned $\texttt{P}_1$ and $\texttt{P}_2$ based on performance and memory consumption on the `LAION2B` dataset. A handful of benchmarks from the search space can be seen in figures 18, 19, 20 & 21.

### 5.5 Search heuristics

Figure 7 depicts an overview of our querying process. The primary goal is to improve the worst-case running time of algorithm 2 by applying a search-heuristics: GetNextBucket, HashMemoization & BatchBucketExtraction.

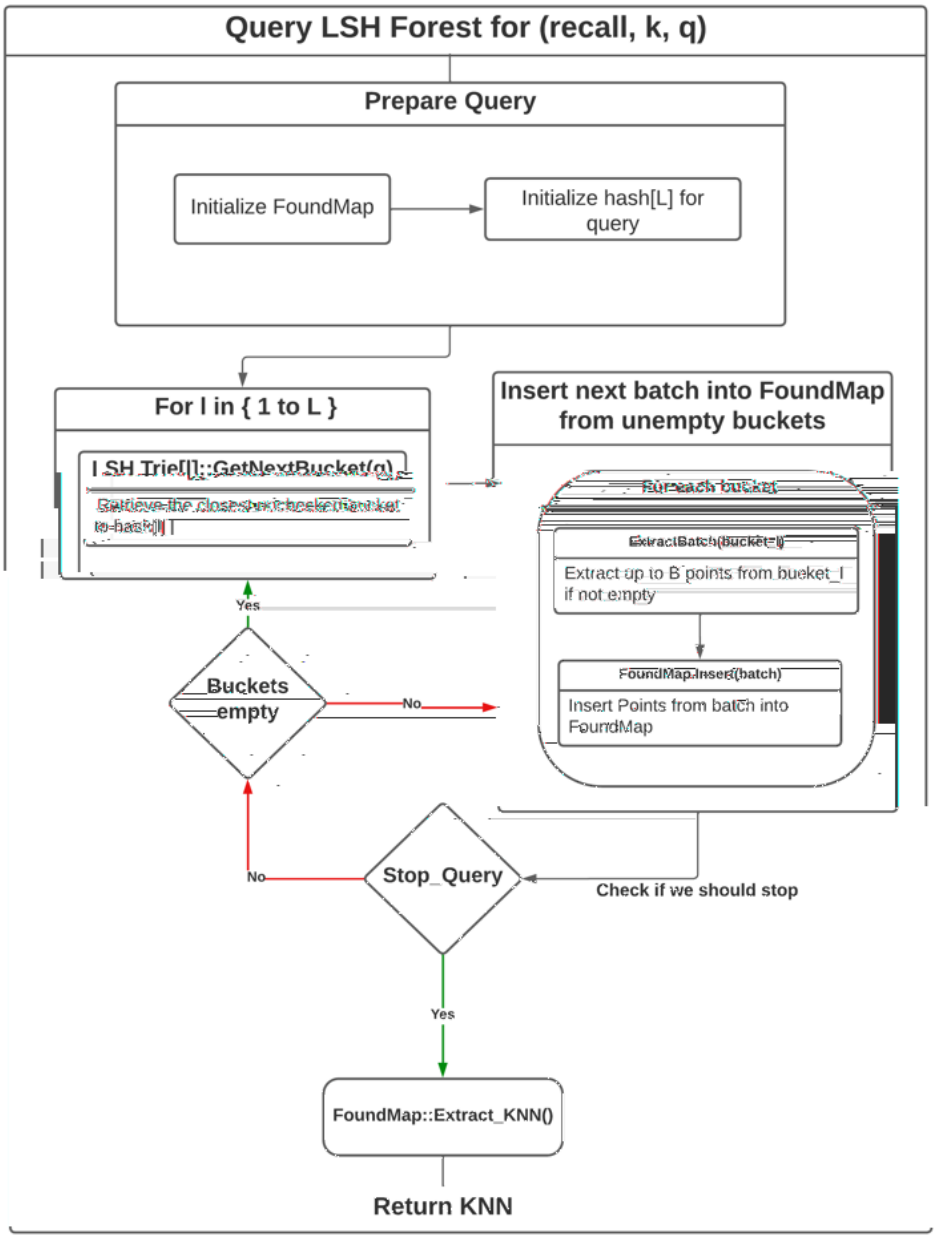


**Figure 7:** *SWANN KNN-QUERY: Overview of how queries are answered through the LSH Forest*

### 5.5.1 Memoize the hash of q for all tries

This optimization aims to reduce the time of GetBuckets to $O(\binom{\mathtt{d}}{\lfloor \mathtt{d}/2 \rfloor})$ by memoizing $\mathtt{S_q}$ for all of the L tries. The initial calculation of $\mathtt{S_q}$ still takes $O(\mathtt{d})$, but each of the up to $\binom{\mathtt{d}}{\lfloor \mathtt{d}/2 \rfloor}$ buckets can be extracted in $O(1)$ time by simply applying the corresponding trie masks (see 5.3). This allows us to improve both the worst-case running time shown in 4 and 5 by removing the d term.

### 5.5.2 GetNextBucket instead of GetBuckets

In algorithm 2 all buckets with exactly hdist hamming distance to hash[l] is extracted from the L LSH-Tries before checking STOP_QUERY$(\lambda(\mathtt{q}, \mathtt{f_k}), \delta, \mathtt{d})$. The GetNextBucket heuristic aim to bound the number of checked buckets between each call of STOP_QUERY$(\lambda(\mathtt{q}, \mathtt{f_k}), \delta, \mathtt{d})$ to L, such that only a single bucket is checked simultaneously for each trie.
While this optimization doesn't improve the theoretical running time (see 5), it improved the average running time of our benchmarks significantly.

### 5.5.3 Extract from buckets in batches

As for GetNextBucket (see 5.5.2), the goal of this optimization is to minimize the batch size $\beta$, where $\beta \quad \mathtt{L}$ denotes the maximum number of candidate points checked between each evaluation of Stop_Query.
Equation 10 shows how to calculate $\beta$ by the multiplication of some bucket factor $\alpha$ and the number of elements queried for k.

$$\beta = \alpha + \mathtt{k}, \quad \text{where } \alpha > 0 \tag{10}$$

**Setting the correct value for $\alpha$** A mathematical proof of the correct value of $\alpha$ is outside the scope of this project, but for Equation 11 illustrates what we used in the implementation.

$$\alpha = \frac{\mathtt{N} \times \delta^{(1-\delta)/0.05-1}}{(1000 + \mathtt{d}) \times \mathtt{L}} + 40 \tag{11}$$

**Advantages** In practise, this optimization made our average running time run about three times faster. Surprisingly, we have also noted that the average number of candidate points checked is always close to $\beta \times \mathtt{L}$.

**Disadvantages** The primary disadvantage of the optimization is that the precision of $\mathtt{P}[F]$ decreases as the probability of false positives increases. To illustrate why, assume L tries, where some trie has a bucket with more than $\beta$ points. After extracting up to $\beta$ points from all the tries, STOP_QUERY evaluates to true, so we stop. Since STOP_QUERY operates under the false

pretence that all of the points in the buckets have been checked, we increase the risk of a false-positive.

**Algorithm 6** (k, $\delta$, q)-NN query with optimizations

**Require:** (k > 0), ($0 < \delta \leq 1$), $\alpha > 0$ and L LSH-Trie as T[1..L]

```
1: F_q ← new FoundMap(P,q,k)
2: hdist ← 0
3: hash ← new int[L]
4: for l ← 1..L do
5:     hash[l] ← T[l].hash(q)
6: end for
7: while stop_query(F_q.get_kth_dist(), δ, d − hdist) do
8:     bucket_q ← new Queue⟨int,int⟩
9:     buckets ← new bucket[L]
10:    for l ← 1...L do
11:        buckets[l] ← T[l].GetBucket(hash[l], hdist)
12:        bucket_q.push(⟨l, 0⟩)
13:    end for
14:    i ← 0
15:    while bucket_q.empty() do
16:        ⟨l, j⟩ ← bucket_q.peek()
17:        bucket_q.pop()
18:        hi ← min(j + α × k, buckets[l].size())
19:        if hi < buckets[l].size() then
20:            bucket_q.push(⟨l,hi⟩)
21:        end if
22:        for z ← j...hi do
23:            F_q.insert(buckets[l][z])
24:        end for
25:        if ++i mod L ≠ 0 then
26:            continue
27:        else if stop_query(F_q.get_kth_dist(), δ, d − hdist) then
28:            return F_q.extract_knn()
29:        end if
30:    end while
31:    if T[0].AllMasksChecked(hdist) then
32:        ++hdist
33:    end if
34: end while
35: return F_q.extract_knn()
```

### 5.6 The optimized algorithm for $(k, \delta, q)$-NN

Algorithm 6 depicts our final ANN query algorithm. The hash memoization optimization presented in 5.5.1 is implemented by lines 3 to 6 by simply precomputing the values of the hashes, and storing them in an array.

To implement the GetBucket implementation from 5.5.2 we have defined a function to help track the number of masks checked per hdist AllMasksChecked. AllMasksChecked will only return true if all the masks for the given hdist have been checked, and is utilized to only increment hdist if there are no more masks (see Table 2).

The batch extraction optimization is implemented by the inner while loop from lines 15 to 29. We use a queue of integer pairs, declared on line 8, to ensure no more than $\beta \times L$ points are added between each evaluation of stop-query.

Each entry $f$l, j$g$ queue holds l, an index of a bucket, and j the index of the first unchecked element in the bucket. The if statement on line 25 intends to reduce the number of stop-query checks by only testing after inserting L batches.

## 6 Methodology

The following section describes our approach to handling and processing data, running benchmarks, and organizing our project solution.

### 6.1 Data preprocessing

The provided datasets for the challenge, which we used to test our project solution, did not contain answers to the nearest neighbours search of the 1024-bit binary sketches. To generate answers for queries, we implemented a brute force algorithm in Python, to find the true nearest neighbours of each query. The brute-force solution computed the distances of $\lambda(q, p_i)$ $8$i $2$ $f1, \cdots, Ng$ and simply returned the k lowest distances found. We deliberately return the distances instead of the indices to account for multiple points having the same distance to q. When calculating the recall of an actual $(k, \delta, q)$-NN, the resulting k point indices would then be transformed into the distances when comparing to the brute-force. These answers were then checked up against, when benchmarking our solution.

### 6.2 LAION-5B dataset

The LAION-5B dataset (Schuhmann et al., 2022) is a publicly available large-scale image-text dataset that contains 5.85 billion image-text pairs. In our

study, we utilized projections of subsets of the dataset into binary sketches, provided by the SISAP comittee (Chavez et al., 2023). These datasets range between 10K to 100M points, and are designed to work with bit-level hamming distance, using 1024 bits.

## 6.3 DevOps

### 6.3.1 Version control

We utilized GitHub as our version control platform to manage the source code of our project, using the `main` branch as the stables branch. Changes to the main branch would be reviewed in pull requests, with GitHub Actions to ensure that all the tests were passing.

The repository can be found at: https://github.com/flopper123/swann

### 6.3.2 Dockerization

To ensure compatibility with latest `Ubuntu Linux` (as per the solution requirements in 2.3), we employed `Docker` for containerization of our solution. It allows us to package our application along with its dependencies into a portable and self-contained container. The necessary steps to build our application image are included in the repository's `Dockerfile`, and the commands to run them are in the `README`.
Additionally, Docker provides an isolated and reproducible run-time environment, ensuring a smoother development environment.

## 6.4 Benchmarks

Throughout the process, we conducted benchmarks to find hyper-parameters and to evaluate the performance and correctness of our solution. As indicated by the work of Aumüller et al. (2020), the bench marking methodology is important, hence to ensure correctness and reproducibility the benchmarks were run locally in Docker containers. Each run of the benchmark generates a `.csv` file containing the results of the benchmark, which is then analysed in a `Jupyter Notebook`.
We used the open source C++benchmark library by Google[11] to run all benchmarks. The library allows for benchmarks to be run with parameters and allows each set of parameters to be run with multiple repetitions to allow for a more accurate result. The code to reproduce the benchmarks is available in on our GitHub repository.
We conducted experiments on a subset of the `LAION2B` dataset consisting of 1024-bit binary sketches (binary vectors), as this is the type of dataset we will encounter in the `SISAP` challenge.

[11] See https://github.com/google/benchmark

# 7 Results

In this section we show the results of our solution on different parameters.

## 7.1 Hyperparameters - P1 & P2

In the appendix figure 18, 19, 20 and 21 shows a subset of our grid searches for hyper-parameters $\texttt{P}_1$ and $\texttt{P}_2$. The bounds of $\texttt{P}_1$ and $\texttt{P}_2$ is selected based on an array of factors and requirements.

1. The average recall within these bounds are above the input recall.
2. The depth of the tries on the largest dataset will remain below the maximum possible depth of 32 (see subsection 5.3).
3. The memory usage is within required bound - both for development and for the 100M challenge (see subsection 2.3).

For the following results, we chose to run the solution with $\texttt{P}_1 = 0.86$ and $\texttt{P}_2 = 0.535$, as these hyper-parameters were the most reliable in terms of recall and good performance with respect to memory footprint.

## 7.2 Bucket Distribution

When analysing our solution, we noticed that the slowest queries were caused by fetching large buckets. As discussed in subsubsection 5.3.2, we began re-building our `LSH`-tries and choosing the `LSH`-trie with the smallest largest bucket.

Figure 8 and figure 9 depicts the average largest bucket of building `LSH`-trie. Optimization step of 1 is equivalent to no optimization, as the optimization steps depicts the number of tries analysed.

Figure 10 and figure 11 shows the average percentage of non-empty bucket, compared to theoretical maximum, alongside the standard deviation. The theoretical maximum would be equivalent of our solution using arrays instead of hash maps. The error bars shows the standard deviation of non-empty buckets.

## 7.3 Candidate points

During benchmarking we collect the average number of candidate points the query has to compute distances for. Figure 12 depicts the average number of candidate points compared to dataset sizes of 100K, 300K and 10M points.

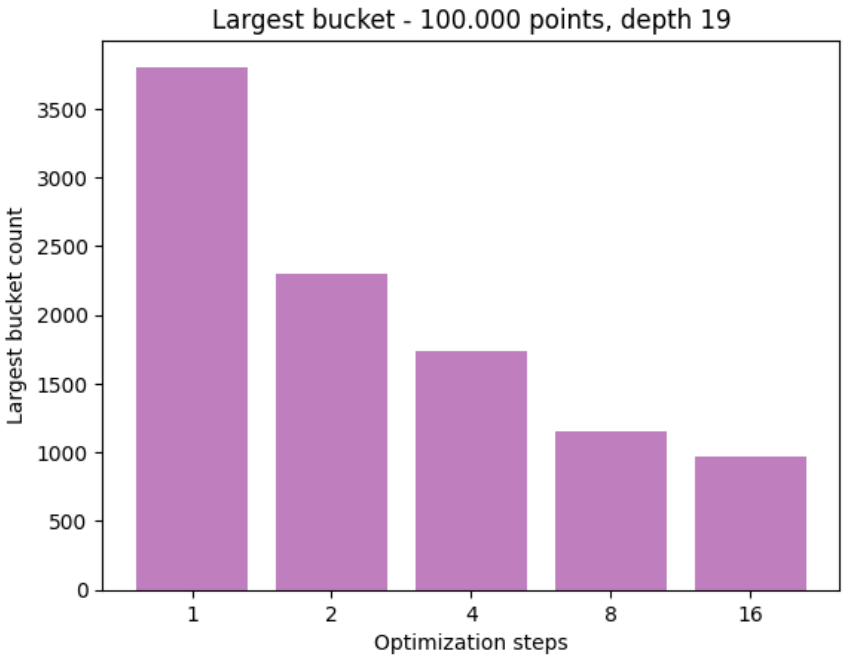


**Figure 8:** *Average largest bucket in LSH-tries (30 repetitions)*

## 7.4 Queries

### 7.4.1 Optimization Iterations

Figure 13 shows the influence of optimization iterations on the average queries per second. The figure only depict the range from 1 to 8 optimization iterations, as the average recall of above 8 optimization steps on 300k points resulted in unstable recalls, where the average recall sometimes would fall below the wanted recall on 10.000 queries.

The build time of the index is proportional to the number of optimization steps, as each trie is rebuild for each step. To ensure the build time of the largest dataset is safely within the 12 hour limit, and to ensure consistent results, the following results have been run with a fixed number of 8 optimization steps.

### 7.4.2 Dataset size

Figure 14 shows the average query time compared to dataset size.

### 7.4.3 Recall

As our solution allows for minimum recall as a parameter, we tested the performance of our solution against different sets of recall values. However,

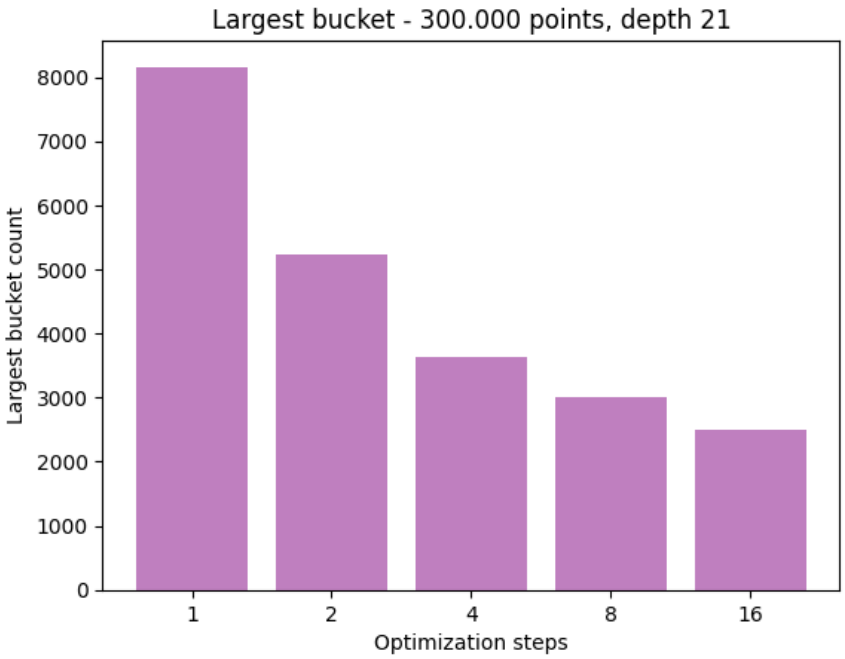


**Figure 9:** *Average largest bucket in LSH-tries (30 repetitions)*

for the SISAP challenge, this value will be fixed to 90%.

- Figure 15 is a subplot of green being input recall and blue being output recall.
- Figure 16 shows the average queries per second compared to recall.
- Figure 17 depicts the average number of candidate points analysed per queries compared to input recall.

# 8 Evaluation

In this section, we evaluate our solution based on the data gathered and visualized in section 7. We evaluate different approaches and parameters based on previously mentioned concepts and optimizations (see section 5).

As many of the input parameters are dependable on each other, testing each parameter in complete isolation is difficult. Running benchmarks on all possible combinations of parameters is also a challenge, as the relative long build times of the indexes is a bottleneck[12]. As such, we have tried to evaluate our solution based on results along with theory, to the best of our ability.

[12]the build time would be near instant if we excluded the trie rebuilding optimization

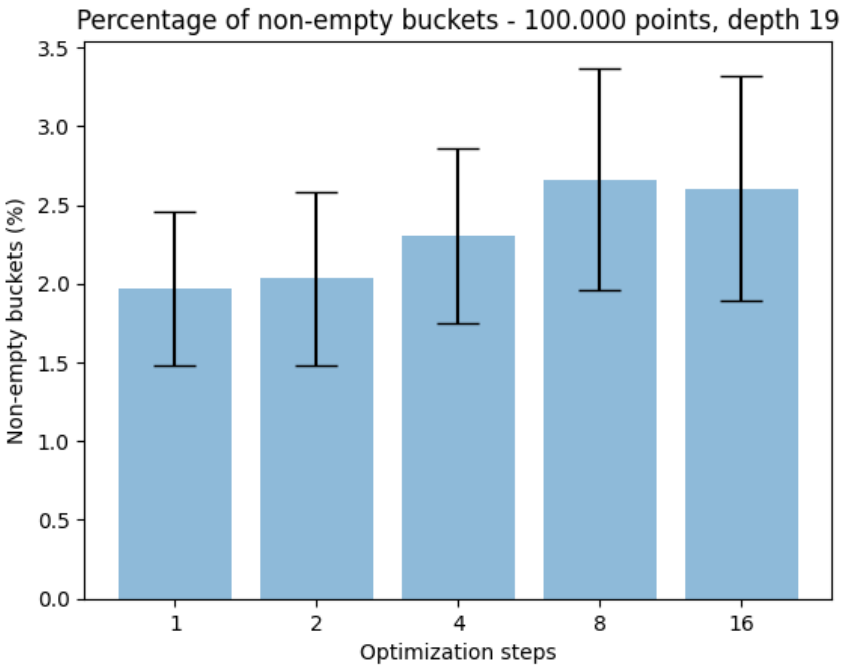


**Figure 10:** *Percentage of non-empty buckets in LSH-tries compared to theoretical maximum of* $2^{depth} = 524288$ *buckets (30 repetitions)*

## 8.1 Hyper-parameters

Our solution requires manual tuning of hyper-parameters. The PUFFINN project is parameterless, as no parameters are needed except index size (Aumüller et al., 2019). Automatic tuning was outside the scope of this project, as this requires a different set of algorithms. The hyper-parameters we searched for are meant to work for the SISAP challenge, hence a completely different dataset might need new parameters to work properly.

**Estimating $P_1$ and $P_2$** As previously explained, figures 18, 19, 20 and 21 depicts a subset from our grid search for parameters $P_1$ and $P_2$. We noticed that most $P_1$ and $P_2$ values close to the shown ranges overall were close to each other in terms of performance. The recall rate near or below the bounds sometimes resulted in an average recall below the input recall. To ensure we stay above the 90% recall requirement of the SISAP challenge (see subsection 2.3), we therefore settled on $P_1 = 0.86$ and $P_2 = 0.535$. This set of parameters consistently gave good results, both in terms of recall and performance, across all of the different dataset sizes.

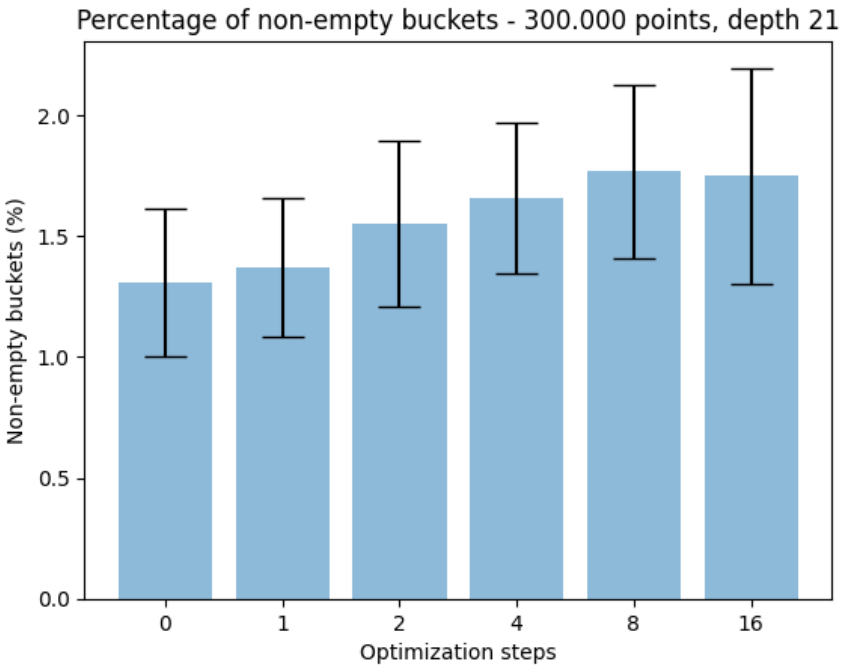


**Figure 11:** *Percentage of non-empty buckets in* `LSH`*-tries compared to theoretical maximum of* $2^{depth} = 2097152$ *buckets (30 repetitions)*

**Optimization steps** Figure 8 and 9 shows the average largest bucket in `LSH`-tries built on `P`, compared to `w` the number of optimization steps. As expected, the average largest bucket size will decrease when the number of optimization steps increases. As different combinations of hash functions result different bucket distributions, rebuilding the tries multiple time should on average allow the tries to find a lower largest buckets. Another approach we attempted was to select the trie with the largest number of non-empty buckets. By using this method, we found that many tries still had large buckets, alongside many buckets only containing a single element, hence being counter productive of our goal with the optimization.

The upward-going trend in figure 10 and 11 is seen as a side effect of our search for the smallest largest bucket. We see a trend that the smaller the largest bucket is, the more non-empty buckets exist. As previously explained, optimization iterations of more than 8 resulted in poor recall on the smaller datasets. As seen from the figures, the average number of non-empty bucket is declining somewhere between 8 and 16 iterations. This could be a symptom of over-fitting where the clustering of the points is no longer based on a random set of hash functions with good distribution, but where the single goal is simply to find a set of hash functions resulting in the smallest

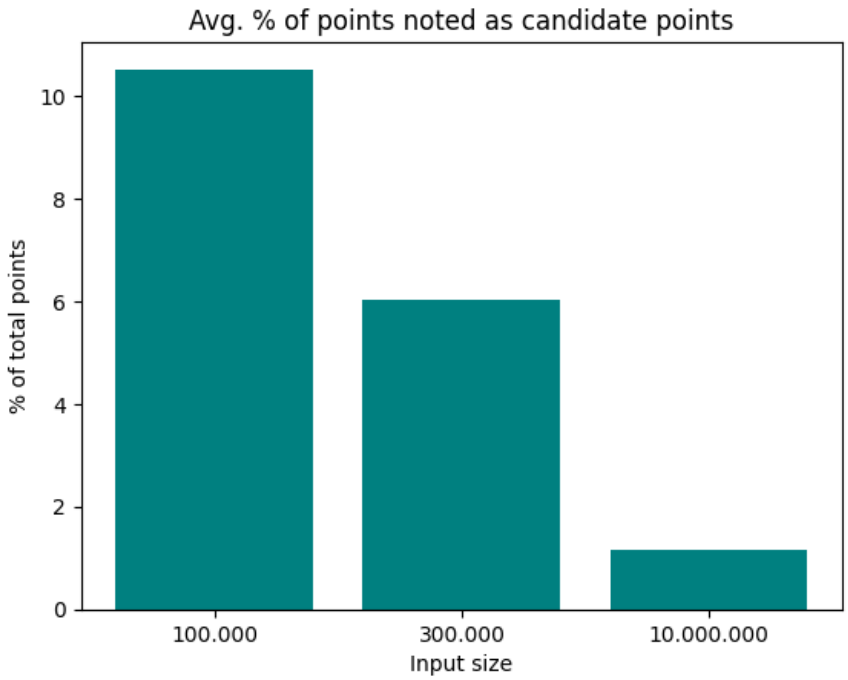


**Figure 12:** *Average percentage of total dataset that are candidate points for each query (3 repetitions)*

largest bucket.

For the dataset of 10M points, we were not able to find a large enough number of iteration steps to cause poor recall[13], it only resulted in faster average queries. However, as expected another issue arose with a large number of optimization iterations - the build time. This is due to the build time increasing linearly with the number of optimization steps (see subsubsection 5.3.2). The build time for 10M points dataset with 8 optimization steps is 30 minutes. For the challenge of 100M points, we wants to ensure that the total time does not exceed the 12 hours, hence we've decided on 8 optimization steps as a fixed size.
As reflected in Figure 13, the optimization iterations greatly improves the average time per query. Other types of datasets may not allow for any optimization iterations. We have theorized that shallow tries may be more susceptible to over-fitting, as there are fewer combinations/permutations of possible single-bit hash functions.

[13] We have tested up to $w = 100$

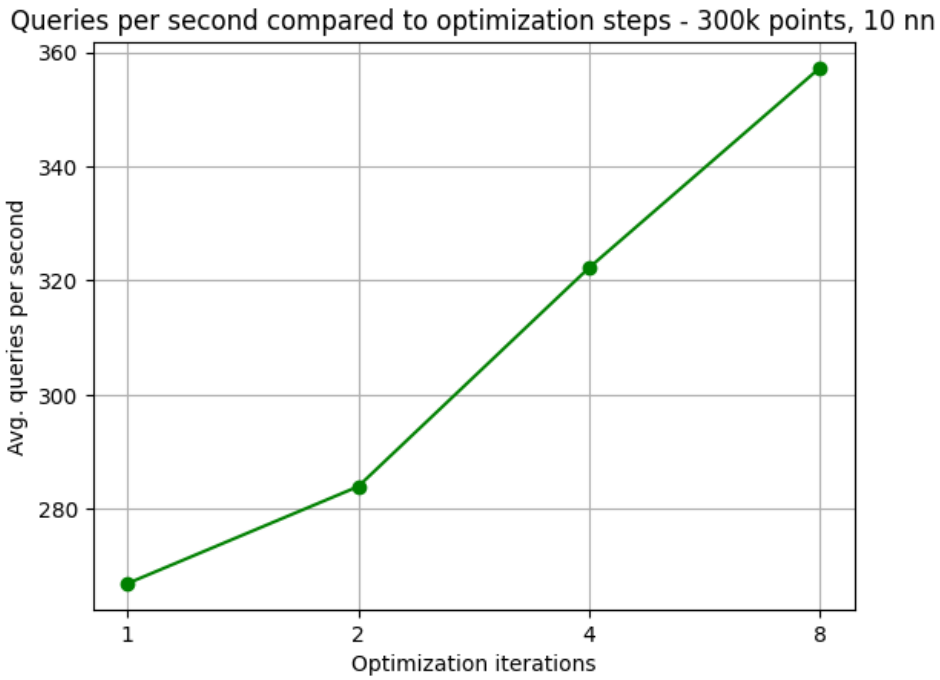


**Figure 13:** *Queries per second compared to bucket optimization iterations (30 repetitions)*

## 8.2 Candidate points

We found our solution on the `SISAP` challenge datasets had sub-linear growth in the average number of candidate per query compared to datasize. From Figure 12, we see for that our solution analyses a smaller fraction of the total dataset the larger the dataset is.
Another factor we expected to influence the fraction of candatidate points was by input recall. In Figure 17, we see the number of candidate points being analysed is super-linear compared to the input recall. This is expected due to the failure probability of the chosen hash family, shown in Equation 8 and Equation 9.

## 8.3 Recall

Though the `SISAP` challenge expects at least 90% recall, we tested the performance on an array of different recall parameter values. From Figure 15 we see that the average actual recall stays above the input recall. We notice that at low input recall values, the actual recall is much greater. From testing, we noticed that this is because a significant fraction of the true `k-NN` were usually be found in the first bucket that each query point hashed to. As a result, the index only has to run a single iteration to find answers to

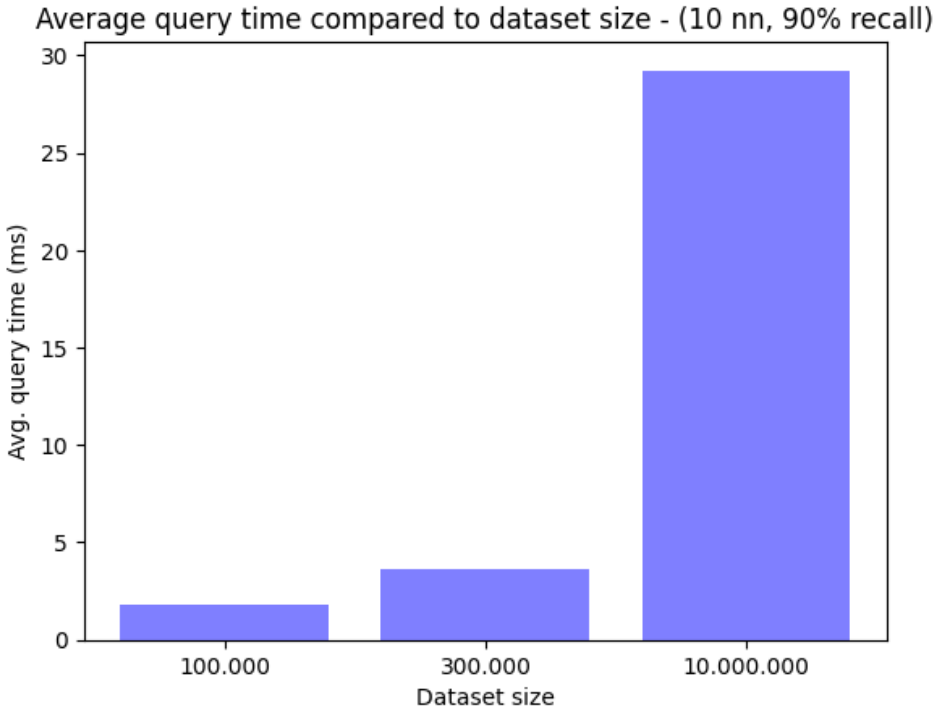


**Figure 14:** *Average query time compared to dataset size (3 repetitions)*

most queries.

## 8.4 Performance

Figure 16 shows how an increase in recall decreases the queries per second. As the number of analysed candidate points grows as recall rate increases, more points on average has to be checked for the failure probability to be low enough to pass the recall threshold.
The number of optimization steps can also be seen influencing the number of queries per second, as shown in Figure 13.
Running benchmarks with $\texttt{w} = 8$ and a recall of 0.90, resulted in Figure 14. The figure shows that even though 300K points is 3% the size of 10M points, our solution only runs 7 times slower on 10M points, which is a sub-linear growth.

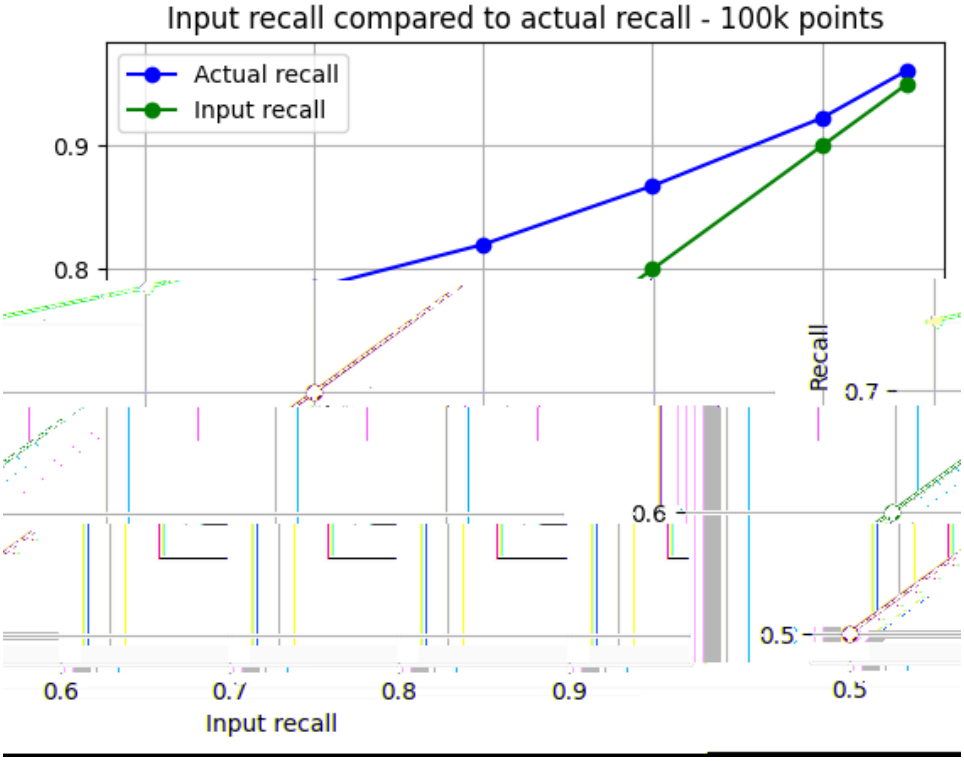


**Figure 15:** *Subplot of actual recall and input recall 100k points (30 repetitions)*

# 9 Discussion

## 9.1 Key differences to `PUFFINN` implementation

**Data structure** For `PUFFINN`, each trie consists of a single array of tuples containing the hash of a point alongside the index to the point. Each array is sorted by the hash value. When querying, the query point is hashed and a binary search is utilized to find the tuple with a hash with the longest common prefix. For each round, the length the prefix has to match is shortened, to evaluate more points.

**Binary Hashfunctions** As mentioned in subsubsection 3.3.2 `PUFFINN` incorporates non-binary hash-functions, and as such their tries are inherently different from `SWANN`'s perfectly balanced tries.

**Sketching** `PUFFINN` implements an optimization strategy called *sketching*, which allows for fewer distance computations based on similarity estimation of hashes between points. Our solution instead reduces distance computations through the FoundMap (see subsection 3.5).

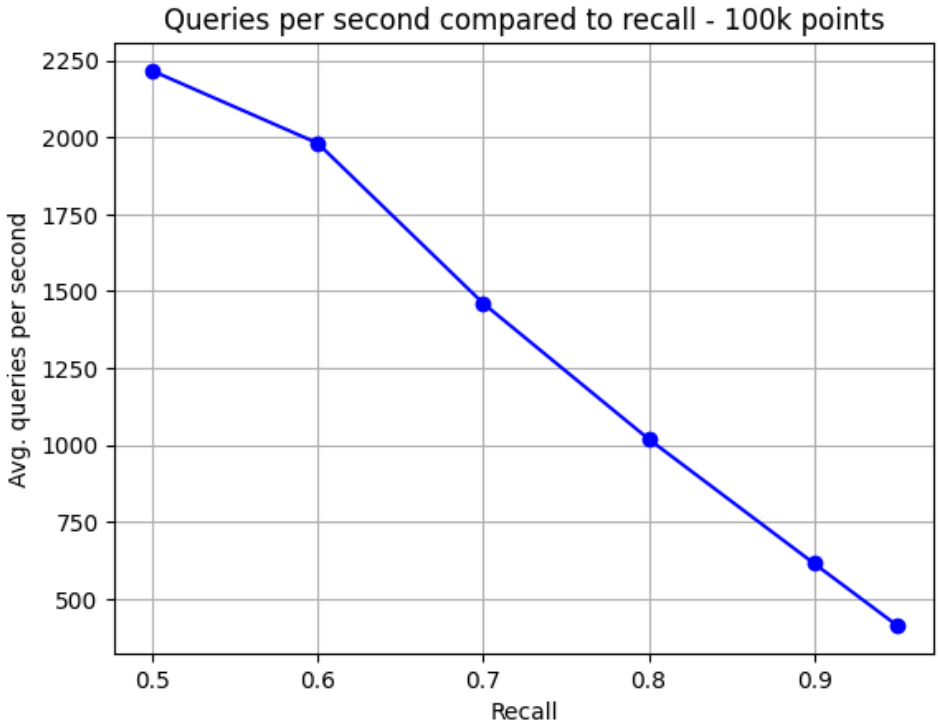


**Figure 16:** *Queries per second compared to recall - searching for 10 nearest neighbours (30 repetitions)*

**Memory consumption** `PUFFINN` allows for setting the memory limit dynamicly through a parameter. `PUFFINN` then proceeds by building its index based on the given amount of memory (Aumüller et al., 2019). Inversely, the memory consumption of `SWANN` is based on the selected hyper-parameters and number of generated buckets.

**Points as bitset** `PUFFINN` offers a flexible solution that works for multiple distance metrics and vector types. In contrast `SWANN`is specialized to only handle binary vectors and only uses the Hamming Distance metric.

**Trie rebuilding** `SWANN`has a data dependent prepossessing step that rebuilds the tries on a given dataset repeatedly to mitigate sub-optimal bucket distribution (see subsubsection 5.3.2).This stands in contrast to the `PUFFINN` implementation that strives to be data independent and therefore only modifies the hash-functions dynamically.

**Extract from buckets in batches** Unlike `PUFFINN`, `SWANN` limits the maximum number of points candidate points extracted before checking for

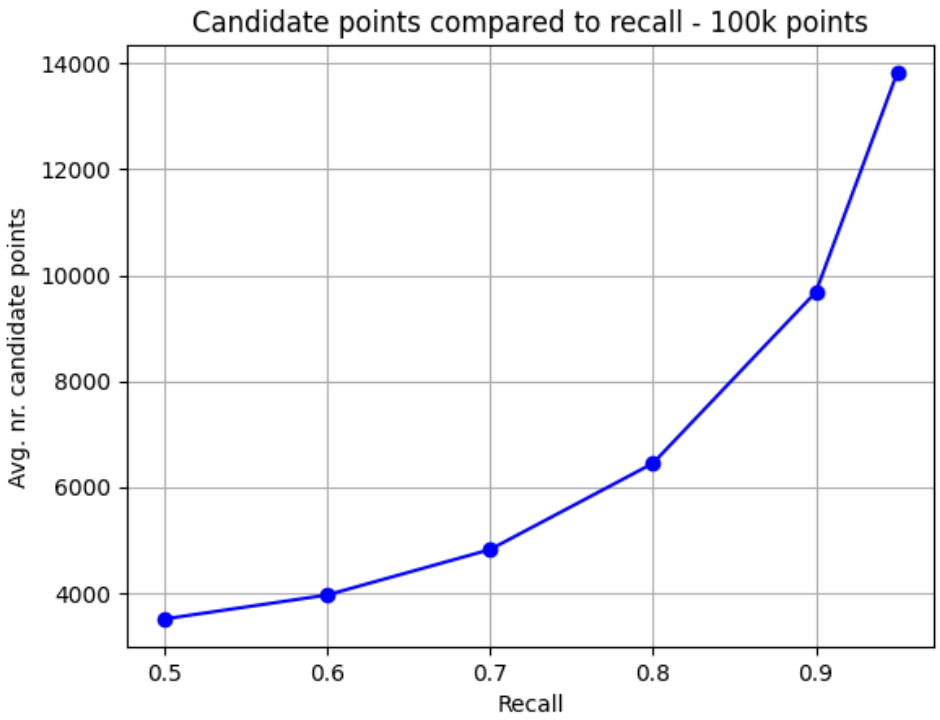


**Figure 17:** *Average number of candidate points to brute-force compared to recall (30 repetitions)*

stop of query through the batch extraction optimization presented in subsubsection 5.5.3.

## 9.2 Future research

As we implemented bucket indices as 32-bit unsigned integers, we are limiting our trie depth to a maximum of 32. We have considered if allowing for deeper trie depths with iterative deepening would result in better performance.
The main idea is to increase the depth at nodes with large clusters, hence creating smaller sub-clusters.
If a node contains small enough clusters, that part of the trie would not become deeper.

We considered running our trie rebuilding in parallel, as this might allow us to increase the number of rebuilds we can perform on large datasets within a given time-frame.

### 9.2.1 Other hashfamilies

Exploring other hash-families is a logical step for future work. A starting point could be the dist_hash function explained below.

**dist_hash** Inspired by the ideas presented by Heo et al. (2012) we researched a promising data dependent LSH-hash family dist_hash. Each member of the hash-family chooses some point a, and some distance threshold $0 < \tau < D$ and applies Equation 12.

$$\text{dist\_hash}(x) = \lambda(\mathtt{x}, \mathtt{a}) > \tau \tag{12}$$

Similar to the rebuild optimization (see subsubsection 5.3.2), each of the trie's hash functions would choose optimal values of a and $\tau$ specifically designed for the dataset P. The bucket distributions of this solution looked very promising, but needs future work, as it's still unclear exactly how to estimate P[F] for this family. Future work could also extend this idea to be non-binary. For example by instead choosing m points and then return the i 2 f1, . . . , mg corresponding to the closest point to x, would get a completely different bucket distribution.

### 9.2.2 Hardware specific optimizations

As mentioned in subsubsection 1.1.1 the SISAP challenge is evaluation is done using a 32-core Intel(R) Xeon(R) CPU E7-4809 workstation. An obvious improvement to tailor our solution to fully utilize the many cores and the optimized Xeon processor.

**AVX-512** The Xeon processor offers the hyper optimized AVX-512 instruction set (Intel, n.d.). Then the hamming distance computation could achieve true constant time evaluation for $D = 1024$ dimensions through the _mm512_popcnt_epi8 (__m512i a) bitcount instruction.

**Multithreaded or parallelism** Due to the many cores of the contest machine, there is potentially a huge gain in multithreading or parallelising the queries. We have actually made multiple attempts at this, but it had little to no effect on our own 8-core machines, and hence decided to leave it out.

# 10 Conclusion

In this paper, we have presented a comprehensive study on our Locality-Sensitive Hashing (LSH) algorithm for indexing and searching high-dimensional binary vectors, building on the work of PUFFINN (Aumüller et al., 2019).

We explored the foundational concepts of LSH, including hash families, bucket distribution, and failure probability. We designed and implemented an LSH-based solution that incorporates trie-based indexing and efficient bucket distribution.

Through benchmarks using the LAION2B dataset, we demonstrated the efficiency and accuracy of our solution. The results showcased the importance of carefully choosing hash families, the impact of hyper-parameters and simple optimizations. We aim to participate in the SISAP contest, and showcase the performance of our solution on Task C of the contest.

Our study contributes to the body of research on LSH algorithms and their potential applications in domains that rely on similarity-based retrieval. We have gained insights into the theory, implementation, and performance of LSH algorithms for efficient indexing and retrieval of high-dimensional binary vectors.

We still see many aspects of the application that can be optimized, both in terms of testing, memory footprint and performance, as this work only scratches the surface of possibilities in LSH.

Due to a mixture of the large room for improvements, and the sub-linear growth represented in our results, we conclude that LSH offers a competitive solution to the ANN problem for binary vectors.

## Acronyms

`ANN` approximate nearest neighbour. 1, 2, 6, 22, 36, 37,

`Big ANN` Billion scale approximate nearest neighbour. 1,

`LAION` Large-scale Artificial Intelligence Open Network. 2, 18, 22, 23, 36, 37,

`LSH` Locality-Sensitive Hashing. 1–10, 12–18, 20, 21, 24–28, 35, 36,

`PUFFINN` Parameterless and Universal Fast FInding of Nearest Neighbors. 1, 2, 4, 7, 27, 32, 33, 36, 37,

`SISAP` Similarity Search and Applications. 1, 3, 11, 17, 23, 26, 27, 30, 35–37

`k-NN` $k$ Nearest Neighbours. 8, 30, 37,

## Glossary

`PUFFINN` Parameterless and Universal Fast FInding of Nearest Neighbors refers to the work of Aumüller et al. (2019). The repository is publicly available on GitHub.

`SISAP`**2023** The `SISAP` 2023 Indexing Challenge is a competition that focuses on evaluating the performance of state-of-the-art methods in similarity search and related applications.

`LAION2B` HDF5 encoded binary vectors of 1024-dimensions extracted from the `LAION-5B` dataset.

**binary functions** Functions for which the output evaluate to either true or false.

**Candidate point** Candidate points defines the set of points who may contain the closest neighbours to a query point.

**Hamming distance** A measure of the number of positions at which two equal-length binary sequences differ. 2

**Recall** The recall of a `k-NN` query is a quality metric denoting the fraction of points reported, that are among the true `k-NN`.

## References

Aumüller, M., Bernhardsson, E., & Faithfull, A. (2020). Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, *87*, 101374. https://doi.org/https://doi.org/10.1016/j.is.2019.02.006

Aumüller, M., Christiani, T., Pagh, R., & Vesterli, M. (2019). PUFFINN: parameterless and universally fast finding of nearest neighbors. In M. A. Bender, O. Svensson, & G. Herman (Eds.), *27th annual european symposium on algorithms, ESA 2019, september 9-11, 2019, munich/garching, germany* (10:1–10:16). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ESA.2019.10

Bawa, M., Condie, T., & Ganesan, P. (2005). LSH forest: Self-tuning indexes for similarity search. In A. Ellis & T. Hagino (Eds.), *Proceedings of the 14th international conference on world wide web, WWW 2005, chiba, japan, may 10-14, 2005* (pp. 651–660). ACM. https://doi.org/10.1145/1060745.1060840

Chavez, E. L., Téllez, E. S., & Aumüller, M. (2023). *Sisap 2023 datasets: 1024-bit binary sketches (hamming)*. Retrieved March 1, 2023, from https://sisap-challenges.github.io/datasets/#1024-bit_binary_sketches_hamming

Datar, M., Immorlica, N., Indyk, P., & Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. Retrieved May 10, 2023, from https://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/p253-datar.pdf

Francisco Santoyo, E. C., & Téllez, E. S. (2014). *Similarity search and applications (SISAP) 7th international conference, Los Cabos, Mexico*. Springer.

Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, & M. L. Brodie (Eds.), *Vldb'99, proceedings of 25th international conference on very large data bases, september 7-10, 1999, edinburgh, scotland, UK* (pp. 518–529). Morgan Kaufmann. http://www.vldb.org/conf/1999/P49.pdf

Heo, J.-P., Lee, Y., He, J., Chang, S.-F., & Yoon, S.-E. (2012). Spherical hashing. *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2957–2964. https://doi.org/10.1109/CVPR.2012.6248024

Intel. (n.d.). *Intel avx-512 - writing packet processing software with intel avx-512 instruction set*. Retrieved May 14, 2023, from https://www.intel.com/content/dam/www/central-libraries/us/en/documents/intel-avx-512-writing-packet-processing-software-with-intel-avx-512-instruction-set-technology-g-1617439595.pdf

Jure Leskovec, J. U., Anand Rajaraman. (2014). Finding similar items. Retrieved May 10, 2023, from http://infolab.stanford.edu/~ullman/mmds/ch3a.pdf

Reece, F. (2019). Normalized flow and sperner theory of coxeter groups. Retrieved May 10, 2023, from https://www.math.uchicago.edu/~may/REU2019/REUPapers/Reece.pdf

Schuhmann, C., Beaumont, R., Vencu, R., Gordon, C., Wightman, R., Cherti, M., Coombes, T., Katta, A., Mullis, C., Wortsman, M., Schramowski, P., Kundurthy, S., Crowson, K., Schmidt, L., Kaczmarczyk, R., & Jitsev, J. (2022). Laion-5b: An open large-scale dataset for training next generation image-text models. Retrieved April 12, 2023, from https://arxiv.org/abs/2210.08402

Simhadri, H. V., Williams, G., Aumüller, M., Babenko, A., Baranchuk, D., Chen, Q., Douze, M., Hosseini, L., Krishnaswamy, R., Srinivasa, G., Subramanya, S. J., & Wang, J. (2021). *Billion-scale approximate nearest neighbor search challenge: Neurips'21 competition track*. Retrieved March 17, 2023, from https://big-ann-benchmarks.com/

Yona, G. (2018). *Fast near-duplicate image search using locality sensitive hashing*. Retrieved March 17, 2023, from https://towardsdatascience.com/fast-near-duplicate-image-search-using-locality-sensitive-hashing-d4c16058efcb

# A Appendix

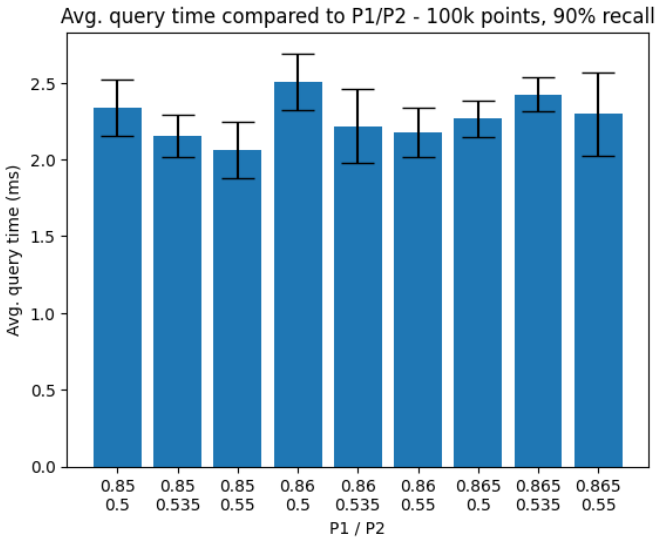


**Figure 18:** *Average query time for 90% recall on 100k points compared to P1 and P2 values (8 repetitions)*

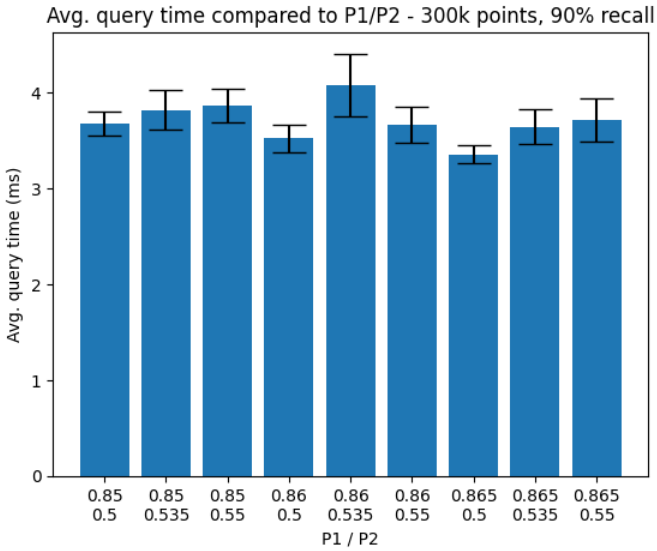


**Figure 20:** *Average query time for 90% recall on 300k points compared to P1 and P2 values (8 repetitions)*

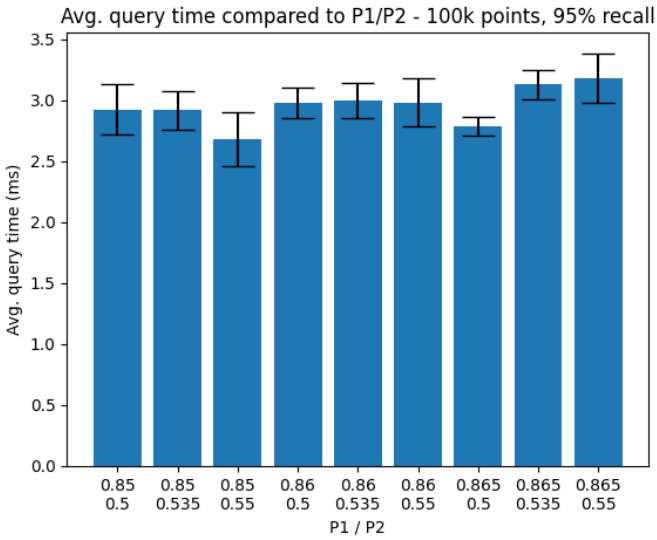


**Figure 19:** *Average query time for 95% recall on 100k points compared to P1 and P2 values (8 repetitions)*

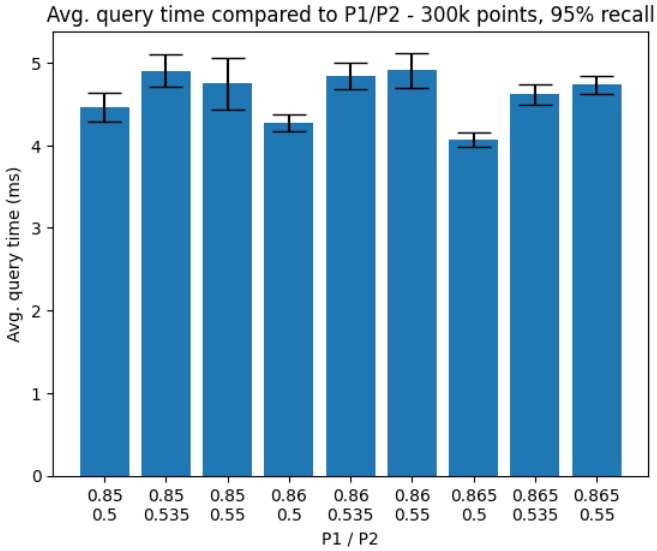


**Figure 21:** *Average query time for 95% recall on 300k points compared to P1 and P2 values (8 repetitions)*