Efficient Closest Pair Identification in Binary Datasets via Multi-Hash LSH

Nick Kulaga

February 26, 2025

Abstract

This study investigates the problem of identifying the closest pair of binary strings in a data set based on Hamming distance. Two approaches are implemented and compared: a brute-force algorithm and a multihash Locality-Sensitive Hashing (LSH) method. Initial explorations with single-level LSH and a Gray code sorting approach were evaluated but found inadequate, prompting a shift to the multihash LSH strategy informed by established literature. Experimental results demonstrate that the multihash LSH method significantly outperforms the brute-force approach in runtime, with a complexity of O(m * n * log(m)) versus O(m^2 * n), where m is the dataset size and n is the string length. The assumption that $n << m$ and its implication that n = O(log(m)) are justified through practical and theoretical considerations, validated by the test results.

1 Introduction

Identification of the closest pair of binary strings, measured by the Hamming distance, is a fundamental problem with applications in error-correcting codes, similarity search, and machine learning. The naive brute-force approach, while accurate, incurs a quadratic time complexity, rendering it impractical for large datasets. Locality-Sensitive Hashing (LSH) offers a probabilistic alternative, leveraging hash functions to group similar items and reduce the search space. This paper presents an implementation and comparative analysis of a brute-force method (bruteForceClosestPair) and a multihash LSH method (multiHashLSHClosestPair), following initial trials with less effective strategies. The study aims to evaluate their performance in terms of run-time and accuracy, with a focus on datasets where string length n is significantly smaller than the number of strings m.

2 Problem Definition

Given a set of m binary strings, each of length n, the objective is to determine the pair with the minimum Hamming distance, the number of positions at which the strings differ. The brute-force method examines all m * (m-1) / 2 pairs, requiring O(m^2 * n) time due to the O(n) cost of computing each distance. In contrast, LSH aims to achieve subquadratic complexity by hashing strings into buckets based on similarity, comparing only within buckets across multiple rounds.

3 Methodology

3.1 Algorithm Implementation

3.1.1 Brute-Force Approach

The baseline algorithm, bruteForceClosestPair, iterates over all pairs of strings, computes their Hamming distance, and returns the pair with the smallest distance:

```
pair<string, string> bruteForceClosestPair(const vector<string>& binaryStrings) {
    int minDist = INT_MAX;
    pair<string, string> closestPair;
    for (size_t i = 0; i < binaryStrings.size(); i++) {
        for (size_t j = i + 1; j < binaryStrings.size(); j++) {
```

```
            int currDist = hammingDistance(binaryStrings[i], binaryStrings[j]);
            if (currDist < minDist) {
                minDist = currDist;
                closestPair = {binaryStrings[i], binaryStrings[j]};
            }
        }
    }
    return closestPair;
}
```

3.1.2 Initial LSH Attempt: Single-Level LSH

An initial LSH implementation, singleLevelLSHClosestPair, employed a single hashing round with k randomly selected bits, followed by pairwise comparisons within buckets and a brute-force fallback if no pairs were identified:

```
pair<string, string> singleLevelLSHClosestPair(vector<string> &vectors) {
    unordered_map<string, vector<string>> buckets;
    size_t minDist = INT_MAX;
    pair<string, string> closestPair;
    size_t k = min(vectors[0].length(), (size_t)log2(vectors.size()));
    for (const string &v : vectors) {
        string key = hashBits(v, k);
        buckets[key].push_back(v);
    }
    // Bucket comparisons and brute-force fallback
}
```

This approach frequently resorted to the fallback due to insufficient bucket collisions, negating its efficiency advantage.

3.1.3 Gray Code Sorting Attempt

A second exploratory method, GrayCodeSortingClosestPair, converted strings to Gray codes, sorted them, and compared adjacent pairs, hypothesizing that minimal distances would appear consecutively:

```
pair<string, string> GrayCodeSortingClosestPair(vector<std::string>& vectors) {
    vector<pair<int, string>> intVectorPairs;
    for (const string &bin : vectors) {
        int grayValue = grayCode(binaryStringToInteger(bin));
        intVectorPairs.push_back({grayValue, bin});
    }
    sort(intVectorPairs.begin(), intVectorPairs.end());
    // Compare adjacent pairs
}
```

This method proved incorrect, as Gray code ordering does not ensure that the globally closest pair is adjacent in sorted order for arbitrary datasets.

3.1.4 Final LSH Implementation: Multi-Hash LSH

After reviewing seminal works by Indyk and Motwani [1] and Gionis et al. [2], which advocate multiple hashing iterations for improved recall, the multiHashLSHClosestPair was adopted. It performs L rounds of hashing with k bits, using parameters scaled with log(m):

```
pair<string, string> multiHashLSHClosestPair(const vector<string>& vectors) {
    const size_t m = vectors.size();
    const size_t n = vectors[0].length();
    const int k = std::max(1, static_cast<int>(std::log2(m)));
    const int L = std::max(1, static_cast<int>(std::log2(m)));
    // Multi-round hashing and bucket processing
}
```

This approach was selected for its balance of efficiency and effectiveness, as supported by the referenced studies.

3.2 Dataset Generation

The generateRandomDataset function generates m unique binary strings of length n using a Bernoulli distribution (p = 0.5). A constraint emerged: for n bits, the maximum number of unique strings is 2^n. Requests exceeding this limit (e.g., m = 500 for n = 8, where $2^8 = 256$) caused infinite loops. To address this, the function caps m at 2^n and enumerates all possible strings from 0 to $2^n - 1$, ensuring at least one pair with a Hamming distance of 1.

3.3 Complexity Analysis

3.3.1 Brute-Force Complexity

The brute-force algorithm evaluates $m*(m-1)/2$ pairs, each requiring O(n) time for Hamming distance computation, yielding a total complexity of $O(m^2 * n)$.

3.3.2 Multi-Hash LSH Complexity

The multiHashLSHClosestPair complexity is derived as follows:

Hashing: $L \approx log(m)$ iterations, each processing m strings with $k \approx log(m)$ bit samples, costing $O(m * log(m))$ per iteration, or $O(m * log^2(m))$ total.

Bucket Processing: Approximately $2^k \approx m$ buckets, with an average size of $b \approx m/2^k \approx 1$. String-to-integer conversion totals $O(m * n)$, and pairwise comparisons sum to $O(m)$ due to random distribution. Per iteration: $O(m * n)$, over L: $O(m * n * log(m))$.

3.3.3 Rationale for $n << m$

The condition $n << m$ aligns with typical high-dimensional datasets where the number of items (m) far exceeds the feature count (n), as in binary feature representations.

4 Experimental Results

The single-level LSH approach exhibited frequent failures, reverting to brute-force due to insufficient bucket population, and was abandoned. The Gray code sorting method consistently failed to identify the globally closest pair, invalidating its use. The multi-hash LSH implementation, however, achieved near-perfect accuracy (after adjusting success criteria to minimum distance equivalence) and significantly reduced runtimes.

Table 1: Runtime Comparisons of Brute-Force and Multi-Hash LSH (Average of 10 Runs)

m (Set Size)	n (Bit Length)	Brute-Force (μs)	Multi-Hash LSH (μs)
50	8	45	61
100	8	252	151
200	8	1127	329
250	8	1735	409
250	12	2418	498
1000	12	38684	2258
4000	12	611268	10666
100	16	523	180
500	16	12687	1175
1000	16	51150	2861
5000	16	1274747	20146
10000	16	5025436	51285
30000	16	45250315	196432
500	20	15297	1268
2500	20	387471	9261
10000	20	6128259	52304

These results highlight LSH's efficiency, especially for larger m.

5 Discussion

The superiority of multiHashLSHClosestPair over bruteForceClosestPair is evident, with $O(m * n * log(m))$ outperforming $O(m * n)$, particularly under the $n << m$ condition prevalent in the test scenarios. The decision to limit testing to $m \leq 2^n$ was justified by the theoretical impossibility of exceeding 2^n unique strings and the irrelevance of such cases to practical applications, as documented in the methodology. The observation that $n = O(log(m))$ aligns with test parameters and enhances the complexity to $O(m*log^2(m))$, though n's contribution remains non-negligible due to string conversions. Literature [1, 2] reinforced the adoption of multi-hash LSH, highlighting the necessity of repeated hashing for robust performance.

6 Conclusion

This investigation demonstrates the efficacy of multi-hash LSH for closest-pair identification in binary string datasets, outperforming brute-force methods in both theoretical complexity and empirical runtime. Initial explorations with single-level LSH and Gray code sorting underscored the importance of robust hashing strategies, guiding the final implementation. The assumptions $n << m$ and $n = O(log(m))$ are substantiated by practical considerations and test outcomes, offering a foundation for future optimizations, such as preprocessing to potentially achieve $O(m * log^2(m))$.

References

[1] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proc. 30th Annu. ACM Symp. Theory Comput. (STOC '98)*, pp. 604–613, 1998.

[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proc. 25th Int. Conf. Very Large Data Bases (VLDB '99)*, pp. 518–529, 1999.