Bachelorproject

Rasmus Løvstad

Block 1+2, 2021

Abstract

This project regards an analysis of recent advances in solving the Approximate Jaccard Similarity Search Problem, specifically in regards to how one can achieve sublinear query time using parallel bit counting as presented by Knudsen[6]. This contains a theoretical analysis of both runtime and correctness of the bit-counting algorithm as well as an empirical comparison to existing methods. The findings include both a theoretical and empirical run time advantage to using parallel bit counting compared to a simple, linear time algorithm. Furthermore, reflections are made on how the results might scale on more specialized hardware.

Contents

1 Introduction

The *Approximate Similarity Search Problem* regards efficiently finding a set A from a corpus $\mathcal{F}$ that is approximately similar to a query set Q in regards to the *Jaccard Similarity* metric $J(A, Q) = \frac{|A \cap Q|}{|A \cup Q|}$[4][6]. Practical applications includes searching through large corpi of high-dimensional text documents like plagiarism-detection or website duplication checking among others[5]. The main bottleneck in this problem is the *curse of dimensionality.* Any trivial algorithm can solve this problem in $O(nd|Q|)$ time, but algorithms that query in linear time to the dimensionality of the corpus scale poorly when working with high-dimensional datasets. Text documents are especially bad in this regard since they often are encoded using w*-shingles* (w contigous words) which Li, Shrivastava, Moore, *et al.* [3] shows easily can reach a dimensionality upwards of $d = 2^{83}$ using just 5-shingles.

The classic solution to this problem is the MinHash algorithm presented by Broder [1] to perform website duplication checking for the AltaVista search engine. It preprocesses the data once using hashing to perform effective querying in $O(n + |Q|)$ time, a significant improvement independent of the dimensionality of the corpus. Many improvements have since been presented to both improve processing time, query time and space efficiency. Notable mentions includes (but are not limited to) *b-bit minwise hashing*[2], *fast similarity sketching*[4] and *parallel bit-counting*[6] (the latter of which is the main focus of this project). These contributions have brought the query time down to sublinear time while keeping a constant error probability.

The addition of parallel bit-counting for querying

2 Theory

2.1 Problem Definition

2.1.1 Jaccard Similarity

2.1.2 Similarity Search Problem

2.1.3 Approximate Similarity Search Problem

2.2 Trivial Solution

2.3 MinHash

2.4 Fast Similarity Sketching

2.5 Parallel Bit Counting

To be able to filter bad matches from good matches, we can use the 1-bit minwise hashing trick to pack the results of multiple subexperiments into one word, which by comparing the bit-wise cardinality (amount of bits set in a bit vector) of two words approximates the similarity between the two. One of the main techniques behind achieving an efficient run time of this method requires computing the cardinality of a bit vector efficiently. Knudsen [6] presents a parallel algorithm to perform this but does neither describe any implementation details and has an unnecessarily complicated proof. I will try to describe the same algorithm in a much simpler fashion and prove its correctness and run time while doing it. A naive algorithm to do this could be described like in algorithm 1.

Algorithm 1 A naive linear time algorithm

```
function CARDINALITY(w, d)    ▷ w is the input word, d is the word-size
    x ← 0
    i ← 0
    while i ≤ d do
        x ← x + (w ≫ i) ∧ 1
        i ← i + 1
    end while
    return x
end function
```

This algorithm trivially runs in linear time to the dimensionality d of the input word w. For n words of size d, this algorithm runs in $O(nd)$ time.

Knudsen [6] presents two improvements to this: The first will improve the run time to $O(n \log(d))$ time by utilizing a divide-and-conquer technique and the second to $O(n + \log(d))$ time by introducing parallelism.

2.5.1 Divide-and-Conquer

To explain how divide-and-conquer methods can be used to improve runtime, we must first introduce a bit mask from [6] defined like so:

$$m_{i,j} = \underbrace{0\ldots0}_{2^j-2^i}\underbrace{1\ldots1}_{2^i}\ldots\underbrace{0\ldots0}_{2^j-2^i}\underbrace{1\ldots1}_{2^i}$$

where $j > i$ and $j, i \in \mathbb{Z}^+$. The notation m_i is a shorthand for $m_{i,i+1}$ and indicates a mask with an equal amount of 1's and 0's. By computing $w \wedge m_{i,j}$ for some word w and some integers i, j, we can isolate a specific segments of size 2^i in the bitstring. We will use this in the following operation:

$$T(w, m, k) = (w \gg k) \wedge m \tag{1}$$

This operation isolates the segments indicated by the bit-mask starting from the kth position of the word w. The algorithm to calculate the cardinality can then be described like in algorithm 2

Algorithm 2 A Divide-and-Conquer approach

```
function CARDINALITY(w, d)    ▷ w is the input word, d is the word-size
    i ← 0
    while i ≤ log_2(d) do
        w ← T(w, m_i, 0) + T(w, m_i, 2^i)
        i ← i + 1
    end while
    return w
end function
```

The simplest way to gain intuition of this algorithm is to prove it. A loop-invariant is presented like so:

Invariant 1. *At the ith iteration of the algorithm, each bitstring-segment of size 2^i of the word w_i will contain the cardinality of the corresponding segment of the word w_0. Furthermore, the operation can be done in constant time.*

Proof. When $i = 0$, then each bit-string of size $2^0 = 1$ in $w_0 = w$ will be 1 if the bit is 1 and 0 if the bit is 0. This proves our base case.

If at step i the word w_i contains $\frac{d}{2^i}$ segments of size 2^i, then invariant 1 will be upheld if we combine each segment with it's neighbor to form a segment of size 2^{i+1}. Since the cardinality of a combined bitstring is equal to the cardinality of its parts, we can divide all of the segments into pairs of size 2^{i+1} and add them together to form the new pair, which is done by the operation $T(w, m_i, 0) + T(w, m_i, 2^i)$ in constant time if the masks are pre-computed. This operation works because $T(w, m_i, 0)$ isolates every other segment of size 2^i starting from the first segment and $T(w, m_i, 2^i)$ isolates every other segment of size 2^i starting from the second segment.

□

When the algorithm terminates after $\log_2(d)$ iterations, the segments will have size $2^{\log_2(d)} = d$ and thus span the entire original word, which means that we have the bit-count of the original word.

2.5.2 Parallelism

To introduce parallelism into the algorithm, we must first realize the following: When two segments of 2^i bits gets combined, they will not need all of the 2^{i+1} bits to represent their sum. It is actually such that the bits used at iteration i is exactly $i + 1$.

Invariant 2. *At the ith iteration of the algorithm, the amount of bits set in a given segment of a word is at most $i + 1$.*

Proof. We will prove this by induction.
At $i = 0$, the size of the segments are $2^0 = 1$, which is equal to $i + 1$.
If it is true at iteration i, then at iteration $i + 1$ the algorithm will add two words of size $i + 1$ which creates a word of size $i + 2$, which fulfills the loop invariant. □

Now, we will introduce a function $l(i)$, which produces the smallest number such that $2^{l(i)} \geq i + 2$. If we use the bit mask $m_{l(i),i+1}$ instead of m_i, we will get the same result. This also means that we can pack $2^{i-l(i)}$ words into one by utilizing the empty space in each segment.

2.6 Comparison

3 Methods

3.1 Hypothesis

3.2 Benchmarking

3.3 Expected Results

4 Implementation

4.1 Technology

4.2 Design

4.3 Assumptions

4.4 Challenges

4.5 Correctness

4.6 Benchmarking

5 Results

6 Discussion

7 Conclusion

References

[1] A. Broder, *On the resemblance and containment of documents*, 1997. DOI: 10.1109/SEQUEN.1997.666900.

[2] P. Li and A. C. König, "Theory and applications of b-bit minwise hashing," *Commun. ACM*, vol. 54, no. 8, pp. 101–109, Aug. 2011, ISSN: 0001-0782. DOI: 10.1145/1978542.1978566. [Online]. Available: https://doi.org/10.1145/1978542.1978566.

[3] P. Li, A. Shrivastava, J. Moore, and A. C. Konig, *Hashing algorithms for large-scale learning*, 2011. arXiv: 1106.0967 [stat.ML].

[4] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, *Fast similarity sketching*, 2017. arXiv: 1704.04370 [cs.DS].

[5] S. Vassilvitskii, *Coms 6998-12: Dealing with massive data (lecture notes, columbia university)*, 2018.

[6] J. B. T. Knudsen, *Fast similarity search with low error probability*, 2021.