CSCI 4114 Final Project

Luke Soguero

December 15, 2020

1 Problem

The problem I will be working on is Closest String. The problem is defined as follows:

> Closest String (CSP)
> *Input:* Strings $s_1, s_2, \ldots, s_k$, all of length l, and some $d \in \mathbb{N}$
> *Decide:* Is there a string s^* such that $\forall i \; Ham(s^*, s_i) \leq d$, where Ham is the function which computes the hamming distance between two strings?

2 Literature

CSP is an important problem in the field of bioinformatics and has many applications in problems that involve finding similar regions of DNA or RNA, such as genetic drug target identification or universal PCR primer design. CSP is known to be NP-complete [1], but there is much literature surrounding heuristic, approximation, and fixed parameter tractable algorithms for CSP. Regarding existing approximation algorithms, it has been shown that CSP can be approximated within a ratio of 2 in polynomial time using a star alignment approximation algorithm [2]. Furthermore, using a more intricate approximation algorithm involving linear relaxations and randomized rounding, CSP can be approximated within a ratio of $\frac{4}{3}(1-\epsilon)$ [2]. CSP has also been shown to be fixed parameter tractable with runtime $O(kn+kd\cdot d^d)$ using a branch and bound algorithm [3]. I will use this algorithm as the basis of my implementation.

3 NP-Completeness

To prove the NP-completeness of CSP it will be shown that CSP is both in NP and NP-hard. Proving that CSP is in NP requires a polynomial time verifier. Given a witness string w to the problem we can verify the problem in $O(k \cdot l)$ time where k is the number of strings in the set and l is the length of each string. A verifier will calculate the Hamming distance between w and each string in the set. Calculating the Hamming distance between one pair of strings takes $O(l)$ time because each pair of corresponding characters is compared. Calculating the Hamming distance k times (w against $s_1, s_2, \ldots, s_k$) results in a runtime of $O(k \cdot l)$. Thus, CSP is in NP. Proving the NP-hardness of CSP is done by reducing 3SAT to CSP. First, consider the case of a 3SAT problem with m clauses and n variables. For each 3-clause c_i over the variables $x_1, ..., x_n$, create a binary vector $\hat{c} = (\hat{c}_1, ..., \hat{c}_{2n})$ such that

$$\forall i = 1...n \quad \hat{c}_{2i-1}\hat{c}_{2i} = \begin{cases} 00 & \text{if } c \text{ contains the literal } \neg x_i \\ 11 & \text{if } c \text{ contains the literal } x_i \\ 01 & \text{otherwise} \end{cases}$$

As an example, the clause $x_1 \vee \neg x_2 \vee \neg x_4$ can be represented as 11 00 01 00 01...01 Now we can define a function Π which "doubles" a vector where $\Pi : \{0,1\}^n \rightarrow \{0,1\}^{2n}$ in such a way that $\Pi(v_1 v_2 ... v_n) = (v_1 v_1 v_2 v_2 ... v_n v_n)$. Now for any binary vector v of length n, $\hat{c}$ is within a Hamming radius of $n+1$ of $\Pi(v)$ iff v is a satisfying assignment for w. We can use this fact to extend our 3-clause case to a full 3-CNF formula. If we create a vector in the way described above for each clause in the formula, we have a set of m binary strings. By solving CSP on this set with $d = n + 1$, we get a binary string $\Pi(v)$ which, when "undoubled", gives us a vector v which satisfies the 3-CNF formula. It takes $O(mn)$ time to create our vectors because for each clause we represent every variable with two bits. Thus, 3SAT is polynomial-time reducible to CSP and we can say that CSP is NP-complete.

4 Fixed Parameter Tractabilty w/ Bounded Search Tree

In their paper "Fixed-Parameter Algorithms for Closest Strings and Related Problems", J. Gramm, R. Niedermeier, and P. Rossmanith outline an algorithm for CSP using branch and bound that has a runtime of $O(kn+kd \cdot d^d)$. We will cover this algorithm in detail and discuss the runtime. First we will

cover how we can preprocess the input to make the computation of our solution easier. One way we can do this is by normalizing the input. When all strings in the input set are stacked on top of one another, they form a 2d matrix. Because Hamming distance is a calculation on the columns of the matrix, the structure of the rows does not matter. We can normalize the input and maintain the structure of the columns by replacing the most frequently occurring character in each column with an 'a', the second most frequently occurring character with a 'b', third with a 'c', and so on. This permutation does not affect the solvability of the problem. Thus, given a solution to a normalized instance, we can find a solution to the original instance by remapping the new characters back to the original characters. An example of normalization is given below:

$$\begin{matrix} x & x & y & w \\ x & z & z & w \\ y & z & w & w \end{matrix} \longrightarrow \begin{matrix} a & b & a & a \\ a & a & b & a \\ b & a & c & a \end{matrix}$$

Next we can reduce our instance to a problem kernel. For a column c_i which only contains one character σ, the optimal solution string will obviously contain σ in position i. These types of columns do not comprise the meat of our problem and can be removed and re-added at the end. Now we are left only with columns containing two or more different characters. If the number of remaining columns exceeds kd then there cannot exist a string s with $max_{i=1,\ldots,k} Ham(s, s_i) \leq d$. The reason for this is that each of the k input strings can differ from the solution in at most d positions. Now we

can use a branch and bound algorithm on our problem kernel.

Algorithm 1: CS(s, d')

Data: Set of strings $s = \{s_1, ..., s_k\}$, integer d
Input: Candidate string s and integer d'
Output: A string $\hat{s}$ with $\max_{i=1...k} Ham(\hat{s}, s_i) \leq d$ and $Ham(\hat{s}, s) \leq d'$

```
if d' ≤ 0 then
    return "not found"
end
if Ham(s, s_i) > d + d' for some i ∈ {1, ..., k} then
    return "not found"
end
if Ham(s, s_i) ≤ d for all i ∈ {1, ..., k} then
    return s
end
Choose some i ∈ {1, ..., k} such that Ham(s, s_i) > d begin
    X = {x|s[x] ≠ s_i[x]}
    Choose some X' ⊆ X w/ |X'| = d + 1
    foreach x ∈ X' do
        s' = s
        s'[x] = s_i[x]
        s_ret = CS(s', d' − 1)
        if s_ret ≠ "notfound" then
            return s_ret
        end
    end
end
```

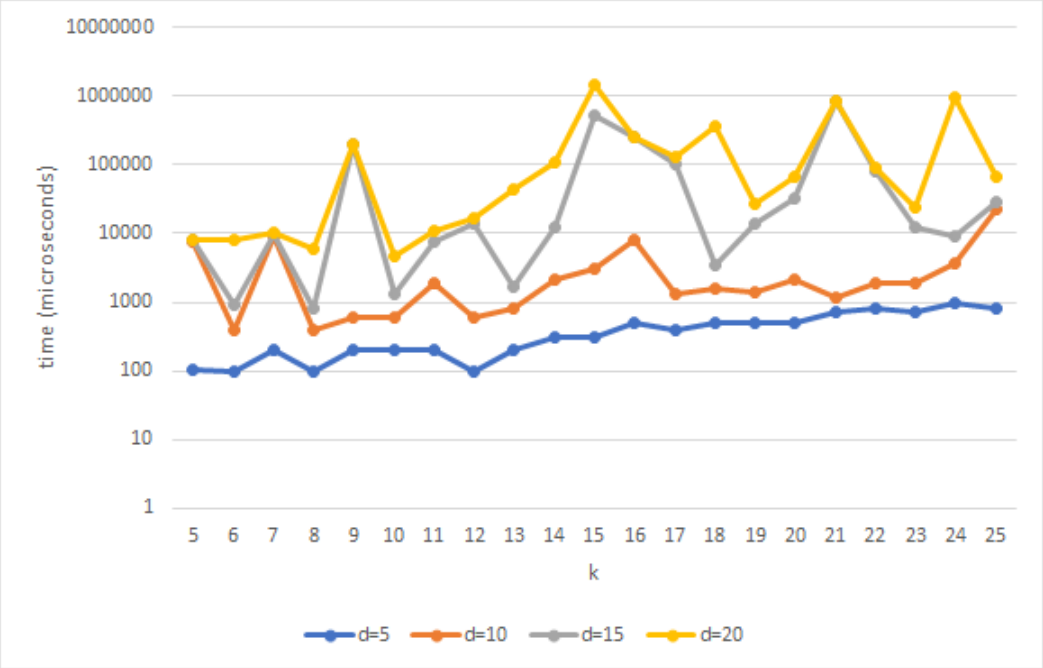
This algorithm relies on the fact that, given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ and integer d, if $Ham(s_i, s_j) > 2d$ for any i and j then there cannot exist a string s with $max_{i=1,...,k} Ham(s, s_i) \leq d$. This is because Hamming distance satisfies the triangle inequality. If $Ham(s_i, s_j) > 2d$ then $Ham(s, s_i) + Ham(s, s_j) > 2d$. Therefore, $Ham(s, s_i) > d$ or $Ham(s, s_j) > d$. If we reach this case in our algorithm, we return "not found". To use the algorithm, we first perform kernelization on the input. We can then check if there are more than kd columns, in which case we reject the instance. Otherwise, we start our recursive procedure with $CS(s_1, d)$. This algorithm essentially starts with a candidate string (in this case s_1) and looks for another string that differs from the candidate in more than d positions. We then recursively make changes to the candidate to move it closer to s_1. We

stop making changes when we have found a solution or when we have moved too far away from s_1. This algorithm always finds a solution if one exists because it recursively tries every change possible to the candidate string. At least one of these changes must be correct for a solution to exist. Now we can calculate a running time for the algorithm. The preprocessing step in which we remove columns with only one character takes kl time because it requires that we check the entire k X l matrix. For the recursion, we can examine our search tree. The algorithm subtracts one from d' on every step of the recursion and terminates when $d' < 0$. We start our recursion with $d' = d$, therefore the height of our search tree is at most d. In each step of the recursion, the algorithm chooses a string s_i and creates a subcase for $d + 1$ positions where s_i and s differ. This gives us an upper bound on the size of the tree of $O((d+1)^d)$. Each step of the recursion must also calculate the Hamming distance between the candidate string and every string in the input set, which takes $O(kd)$ time due to fact that each string has at most d characters after preprocessing. Our final runtime is $O(kn + kd \cdot d^d)$, making CSP FPT with respect to parameter d.

5 Implementation

I implemented the algorithm described above in C++. I applied an additional heuristic which chooses an ideal starting string rather than simply starting with s_1. To do this, the algorithm finds the string in the input set with the lowest average distance to all other strings. This way, the algorithm will need to explore less of the search tree on average. To create random instances to test on, I start with a random string of length l over an alphabet of size 4. Then I create k strings which differ from the initial string in d random positions.

6 Results



The test were performed on random instances with a constant $l = 50$. The lower d values ($d = 5$ and $d = 10$) show a linear increase in runtime as k increases, which is expected. On higher d values, the trend becomes much more jagged. I speculate that this may be due to some issue with the way that the random instances are being generated, but I am unsure. As expected, an increase in d results in an exponential increase on the runtime.

References

[1] M. Frances and A. Litman. On covering problems of codes. Theory of Computing Systems, 30:113–119, 1997.

[2] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In Proceedings of the 10th ACM–SIAM Symposium on Discrete Algorithms (SODA), pages 633–642.

[3] J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for closest string and related prob-lems.Algorithmica, 37(1):25–42, 2008

[4] Chimani, Markus Woste, Matthias Böcker, Sebastian. (2011). A Closer Look at the Closest String and Closest Substring Problem. 2011 Proceedings of the 13th Workshop on Algorithm Engineering and Experiments, ALENEX 2011. 13-24. 10.1137/1.9781611972917.2.