Playing with Cyclic Edit Distance and Planar Graphs

Christian Howard
howard28@illinois.edu

Abstract

In this report, we discuss a generalization of the edit distance problem to allowing for cyclic strings, a problem we call the *Cyclic Edit Distance Problem*. The cyclic edit distance problem considers two strings s_1 and s_2 and aims to find the minimal edit distance of these strings across all cyclic shifts of the strings. We discuss some of the literature on this problem and present a new $O(mn)$ time algorithm based on planar maps and duality that ultimately matches the asymptotic runtime of the classic edit distance problem. Finally, we present experimental results contrasting the runtime performance of the new algorithm relative to the historical techniques.

Contents

1 Introduction

Since the early days of computer science, being able to perform efficient computations and manipulations with strings has been of exceptional importance. What began with problems such as identifying whether some string s_1 is contained within a string s_2 has grown to large scale string related problems found in computational biology, machine learning, databases, and more [1, 2, 9, 16–19, 21]. Indeed performing efficient computations on strings is an area that garners much attention as the number of applications for such algorithms are extensive. Dating back to the 1960s in Russia [12], Levenshtein defined what many refer to as the *edit distance* between two strings. Thanks to this metric, one of the early problems that is now taught in many introductory algorithms courses is that of the *Edit Distance* problem.

Problem 1 (Edit Distance) *Given an alphabet Σ and two strings $s_1 \in \Sigma^n, s_2 \in \Sigma^m$ where $m \leqslant n$, compute the minimum number of character removals, additions, or substitutions needed to transform string s_2 into s_1. This distance, which will be written as $d_{edit}(s_1, s_2)$, is called the Edit Distance.*

As a brief refresher, a naive approach to the Edit Distance problem described above can result in exponential run time that would seemingly make this problem appear quite hard. However, With a small change in perspective, this problem described above has a simple and elegant solution that can be obtained by use of dynamic programming [6, 7]. More specifically, the resulting dynamic programming algorithm computes the edit distance for two strings in $O(mn)$ time and can be made to use as small as $O(m)$ extra numbers to obtain the solution, in turn requiring at most $O(m \log(m) + n \log |\Sigma|)$ bits of space in total to store the input strings and the extra space. While the edit distance problem has been demonstrated to have an exact algorithm with relatively small run time and good memory usage, what happens if we generalize this problem to find the minimum edit distance between all cyclic shifts of s_2 and the string s_1? This problem, which we will refer to as the *Cyclic Edit Distance*, is described below.

Problem 2 (Cyclic Edit Distance) *Given two strings s_1, s_2 of length n and m respectively, compute the minimum number of character removals, additions, or substitutions needed to transform any cyclic shift of s_2 into s_1. For some string A, let $A^{(k)}$ correspond to a k-shift cyclic permutation of A. The Cyclic Edit Distance of strings s_1 and s_2, written as $d_{cyc\text{-}edit}(s_1, s_2)$, can be written compactly as*

$$d_{cyc\text{-}edit}(s_1, s_2) = \min_{k \in \{0, \cdots, m-1\}} d_{edit}(s_1, s_2^{(k)})$$

In this report, we cover some of the historical work done on tackling the cyclic edit distance problem and present a new $O(mn)$ time algorithm that makes use of planar maps and duality to reduce the cyclic edit distance problem to an equivalent flow problem that can benefit from some fast algorithms. We then present the experimental results of the historical techniques for computing the cyclic edit distance and contrast them to the new method described.

2 Prior and Related Work

2.1 The Edit Distance Problem

We begin our discussion of prior work by first going over the key details behind how one approaches efficiently computing what we call the *Edit Distance*, as defined in Problem 1. Going off of that problem description, it can be shown that a naive approach to this problem produces a exponential run time that would seemingly make this problem appear quite hard. However, with a small change in perspective, a simple and elegant solution that makes use of dynamic programming emerges, an approach which was independently discovered, sometimes by tackling slightly different but roughly equivalent problems, by numerous researchers in the 1960s and 1970s [20, 23–25]. More specifically, the resulting dynamic programming algorithm computes the edit distance for two strings in $O(mn)$ time and can be made to use as small as $O(m)$ extra counters to obtain the solution, in turn requiring at most $O(m \log(m) + n \log |\Sigma|)$ bits of space in total to store the input strings and the extra space. The key insight for this problem is to first obtain a recurrence for the edit distance value and then visualize this recurrence as a graph. Given we describe $d(i, j)$ as the edit distance between strings $s_1[1, \cdots, i]$ and $s_2[1, \cdots, j]$, the recurrence we wish to find can be described as

$$d(i,j) = \begin{cases} 0 & (i,j) = (0,0) \\ \min \begin{cases} 1 + d(i-1, j) \\ 1 + d(i, j-1) \\ \delta_{ij} + d(i-1, j-1) \end{cases} & (i,j) \in ([n] \times [m]) \setminus \{(0,0)\} \\ \infty & i < 0 \text{ or } j < 0 \end{cases} \tag{1}$$

where $\delta_{ij} = 1$ if $s_1[i] \neq s_2[j]$ and is 0 otherwise. Note that the minimum operation inside the recurrence corresponds to choosing whether it is best to insert the i^{th} character of s_1 before $s_2[j]$, delete the j^{th} character in s_2, substitute the j^{th} character of s_2 with the i^{th} character of s_1, or skip both the characters from the two strings because they have the same character value. If we define $[p] := \{0, 1, \cdots, p\}$, then this recurrence corresponds to a state-space directed graph,

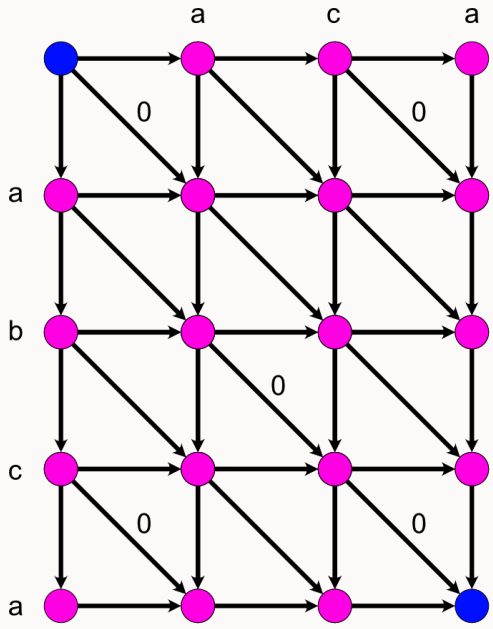

Figure 1: Edit distance DAG with strings $s_1 = abca$ and $s_2 = cab$. All edges have weight 1 unless labeled explicitly with a 0 weight value

where a given vertex in this graph is an (i, j) pair from the set of vertices $V = [n] \times [m]$. We can then draw an edge from vertex $u = (i_u, j_u)$ to vertex $v = (i_v, j_v)$ if $d(i_v, j_v)$ depends on $d(i_u, i_u)$ in the recurrence formulation. If indeed there is an edge $u \to v$ in this graph, we then assign a weight of $w(u \to v) \in \{0, 1\}$ to this edge depending on whatever weight $d(i_u, j_u)$ is associated with in the recurrence. If we then draw this state-space graph onto a 2-dimensional grid, we can observe this graph is a planar directed acyclic map with edge weights in $\{0, 1\}$, with an example of its structure shown in Figure 1.

		a	*b*	*c*	*c*	*a*	*b*	*b*	
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
a	1	1	0	1	2	3	4	5	6
c	2								← current
a	3								
b	4								

Figure 2: Edit distance, row by row

As our goal is to ultimately compute $d_{\text{edit}}(s_1, s_2) = d(n, m)$, we can use the grid structure of the planar DAM to compute the desired edit distance by first representing the map as a matrix

and, say, starting at the top boundary where we have known boundary conditions and then step down one row at a time. We can then compute the intermediate edit distance values at each vertex on a fixed row by using the value in the row before it, as shown in Figure 2. As we only need one row of information to compute the row below it, this implies we can work our way down to the bottom row where $d(n, m)$ is by using at most $O(m)$ space. We can alternatively do this working from the left boundary to the right one in a similar manner, using at most $O(n)$ space. Since we assume that $m \leqslant n$ for sake of discussion, choosing to start from the top boundary gives us an $O(mn)$ time algorithm that uses at most $O(m)$ extra counters worth of space, implying a total space of $O(m \log(m) + n \log |\Sigma|)$.

2.2 The Cyclic Edit Distance Problem

2.2.1 The Naive Algorithm

Now the edit distance problem has been demonstrated to have an exact algorithm with relatively small run time and good memory usage, so what happens if we generalize this problem to find the minimum edit distance between all cyclic shifts of s_2 and the string s_1? This problem, which we will refer to as the *Cyclic Edit Distance*, was described earlier in Problem 2. Using the simple algorithm for edit distance, we immediately obtain a $O(m^2 n)$ algorithm with a total space usage of $O(m \log(m) + n \log |\Sigma|)$ bits for computing the cyclic edit distance of two strings s_1 and s_2. This can be seen by computing the edit distance between s_1 and each of the m cyclic shift permutations of s_2 and just returning the minimum value seen. While this is indeed simple and low on memory, the run time is likely to be prohibitively expensive in a serial environment with input strings that are not small. So what might we do to achieve an improvement?

2.2.2 A Divide and Conquer Algorithm

In 1990, Maurice Maes took up the mantle in investigating an improvement to the cyclic edit distance problem with more general costs for the edit operations, something he called the *Cyclic String-to-String Correction* problem [14]. Maes made a key insight by observing that one can view the cyclic edit distance problem as a multiple source shortest path (MSSP) problem using a collection (s, t) vertex pairs in a state-space graph that corresponds to the edit distance graph between the string s_1 and the concatenated string $s_2 s_2$. If vertices of this graph are embedded into the plane by defining the vertices as $V = [n] \times [2m - 1]$ and the edges follow from the edit distance recursion, as before, then the (s, t) vertex pairs we will perform shortest path computations on correspond to $s_i = (0, i)$ and $t_i = (n, i + m)$, for all $i \in [m - 1]$. For the weights in our graph, the shortest path from any start vertex s_i to its corresponding end vertex t_i can be computed using a breadth-first search (BFS) based algorithm..

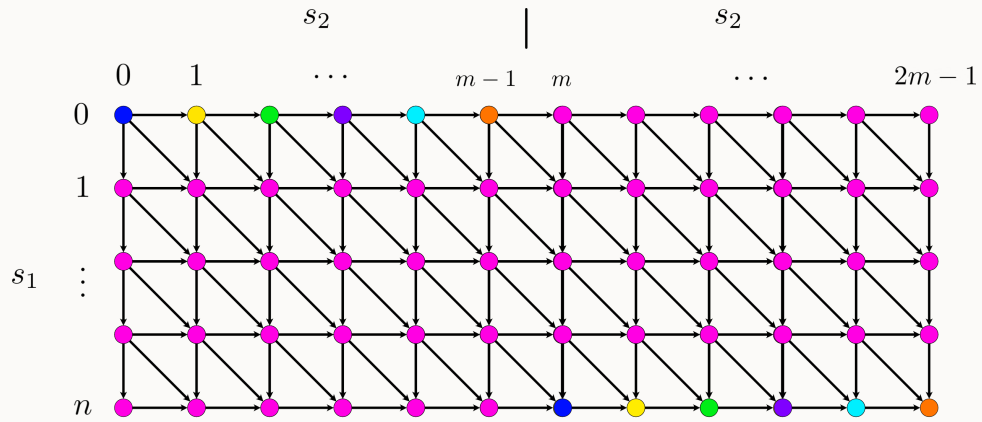

Figure 3: Multiple source shortest path (MSSP) formulation for cyclic edit distance, where (s, t) vertex pairs share same colors.

Lemma 2.1 (Maes (1990)[14]) *The cyclic edit distance problem can be solved exactly in* $O(nm\log(m))$ *time and naively* $O(mn\log(m))$ *extra space.*

With the above in mind, Maes was able to show the result described in Lemma 2.1 using a divide and conquer algorithm that is described in Algorithm 1. The idea here is to initially compute the shortest paths P_0 and P_{m-1} for the vertex pairs (s_0, t_0) and (s_{m-1}, t_{m-1}), respectively, to help reduce the number of vertices the other (s, t) vertices would need to go through to perform their shortest path computations. For $i < k$, we apply an iterative realization of a recursive algorithm where, given shortest paths P_i and P_k for pairs (s_i, t_i) and (s_k, t_k), respectively, we compute the shortest path P_α between the median pair (s_α, t_α), where $\alpha = \lfloor\frac{i+k}{2}\rfloor$. Abstractly, we call this recursive algorithm once on the tuple (P_i, P_α) and once on the tuple (P_α, P_k), returning the smallest path distance found in both recursive sub-problems. As the algorithm is written, it terminates after a shortest path P_i is found for each $i \in [m-1]$ and the algorithm returns the smallest path distance seen.

The runtime is dominated by the shortest path computations. If we fix the recursion depth j, any vertex can be found to show up in at most a constant number of shortest path computations. We can get some sense of this by looking at Figures 4 through 7. Since the depth of the recursion tree is $O(\log(m))$, this implies the runtime is at most $O(mn\log(m))$ with a naive space use of $O(mn\log(n) + n\log|\Sigma|)$ total bits with the possibility of improving to at most $O(n\log(n)^2 + n\log|\Sigma|)$ total bits of space. Thus, Maes' algorithm gives us one of our first non-trivial results.

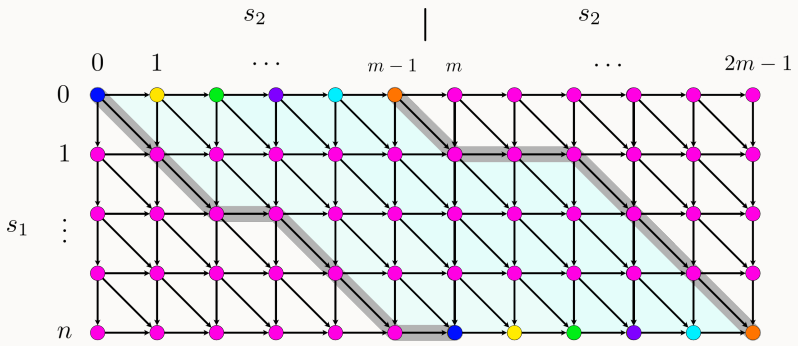

Figure 4: Maes' algorithm after computing P_0 and P_{m-1}

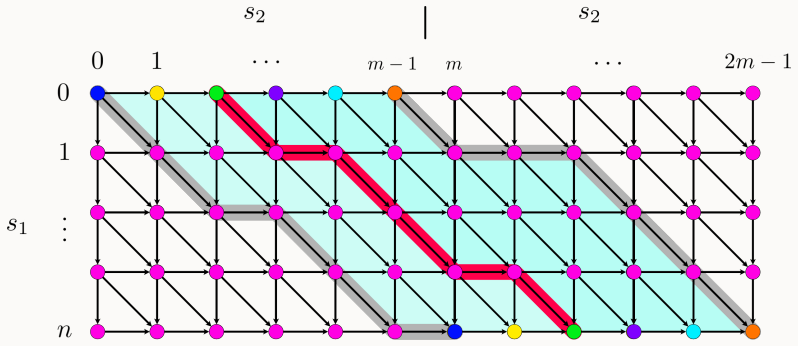

Figure 5: Maes' algorithm for $j = 1$ and $k = 1$

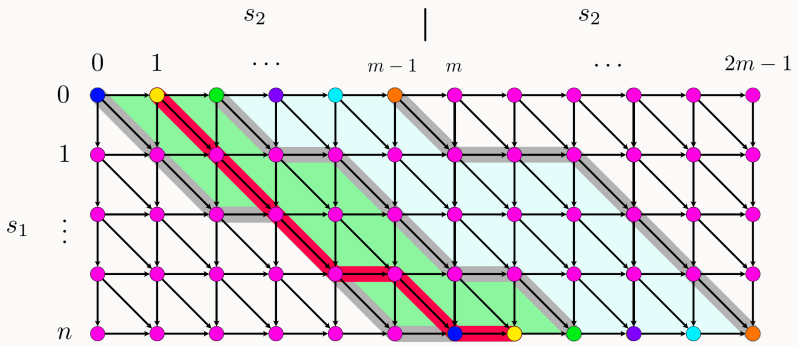

Figure 6: Maes' algorithm for $j = 2$ and $k = 1$

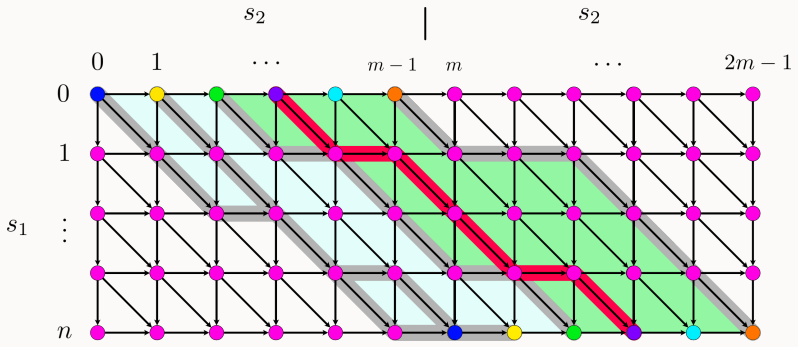

Figure 7: Maes algorithm for $j = 2$ an $k = 2$

Algorithm 1 Maes' Iterative Divide and Conquer Cyclic Edit Distance

procedure MAES-CYCLIC-EDIT-DISTANCE(s_1, s_2)
    Compute shortest paths P_0 and P_{m-1}
    $q \leftarrow \log_2(m)$
    $d_{\min} \leftarrow \infty$
    for $j = 1, 2, \cdots, q$ **do**
        for $k = 1, 2, \cdots, 2^{j-1}$ **do**
            Compute a minimum weighted path $P_{(2k-1)(m-1)/2^j}$ between paths
            $P_{(2k-2)(m-1)/2^j}$ and $P_{2k(m-1)/2^j}$ using a modified edit distance algorithm and
            store path weight $\hat{d}$
            Update $d_{\min} \leftarrow \min\{d_{\min}, \hat{d}\}$
        end for
    end for
    return $d_{\min}$
end procedure

2.2.3 An Incremental Algorithm

In 2004, a work by Sung-Ryul Kim and Kunsoo Park presented a relatively simple dynamic solution for tackling the edit distance problem [11] with properties given by Lemma 2.2.

Lemma 2.2 (Kim & Park (2004)[11]) *There exists a data structure D for incremental edit distance using $O(mn)$ bits of space that allows for:*

- *Initializing with two strings s_1 and s_2 in $O(mn)$ time*
- *Updating in $O(n+m)$ time input string s_2 by character removals or additions at the front or back of the string*
- *Computing edit distance value in $O(n+m)$ time*

With the high level details of the data structure in mind, one can see by Algorithm 2 that we are able to compute the cyclic edit distance in $O(mn)$ time in what turns out to be at most $O(mn + n\log|\Sigma|)$ bits of memory. The initialization phase requires $O(mn)$ time and then each iteration of the loop requires $O(n+m)$ time based on Lemma 2.2. Since the loop is executed $O(m)$ times, this implies a total runtime of at most $O(mn)$, as expected.

Algorithm 2 Kim & Park Cyclic Edit Distance

procedure KIM-PARK-CYCLIC-EDIT-DISTANCE(s_1, s_2)
    Build initial data structure D using s_1 and s_2
    $d_{\min} \leftarrow D$.compute-edit-distance()
    for $j = 1, 2, \cdots, m-1$ **do**
        Remove front character c from s_2 and update D
        Add character c to back of s_2 and update D
        $d_{\min} \leftarrow \min\{d_{\min}, D$.compute-edit-distance()$\}$
    end for
    return $d_{\min}$
end procedure

The main technique the Kim and Park use to arrive at the dynamic edit distance data structure and corresponding algorithm involves finding a nice way to represent what they refer to as a *D-Table*, which is equivalent to a matrix of edit distance values where row i and column j corresponds to $d(i, j)$ from the recurrence described in (1). What they do is represent this D-table using a difference representation where each (i, j) indexing corresponds to 3 bits, as the edges of our edit distance graph are either 0 or 1. So given that D is the D-table matrix, the three bits corresponding to the difference representation at a position (i, j) is defined as

$$DR(i,j).U = D(i,j) - D(i-1,j)$$
$$DR(i,j).L = D(i,j) - D(i,j-1)$$
$$DR(i,j).UL = D(i,j) - D(i-1,j-1)$$

Park and Kim define what they call a *change table* Ch that represents the change between some D-table D to what it becomes after an update, D'. Using the change representations for D and D', DR and DR' respectively, the authors are able to work out an algorithm that turns DR into DR' by first identifying that the change table has structure where the grid of values are partitioned into three connected regions: one that contains values of -1, one that contains values of 0, and another that contains values of 1. They go on to observe that DR only changes along two boundaries, namely the boundary between the (-1)-region and the 0-region and the boundary between the 0-region and the 1-region. By some careful case analysis, Park and Kim give an algorithm for sweeping through the two boundaries efficiently, identifying what difference table values in DR we need to change, and updating them so that DR becomes DR', all in $O(n + m)$ time.

3 Cyclic Edit Distance with Planar Maps

In this section, we will go over an exact $O(mn)$ time algorithm proposed by my advisor Jeff Erickson that he and I have been doing some work on that uses ideas from planar graphs and duality.

3.1 Duality for the win

In this section, we begin by discussing a sequence of reductions from our initial cyclic edit distance problem that allow us to benefit from some other fast algorithms. In particular, the algorithm we will explain proves the following

Theorem 3.1 *Given a language Σ and strings $s_1 \in \Sigma^n, s_2 \in \Sigma^2$ with $m \leqslant n$, we can compute $d_{\text{cyc-edit}}(s_1, s_2)$ exactly in time $O(mn)$ and $O(mn\log(n) + n\log|\Sigma|)$ bits of space using an algorithm based on planar maps and duality.*

To prove this theorem, we will specify and prove or cite a variety of useful lemmas.

Lemma 3.2 (Directed to Undirected) *Given a directed acyclic planar map $\vec{G}$ constructed from a cyclic edit distance input instance, every directed edge can be replaced by an undirected edge and still maintain the same shortest paths found in $\vec{G}$ between each (s_i, t_i) pair.*

Proof. Suppose this claim is not true and let G be the planar map made by replacing each directed edge of $\vec{G}$ with an undirected edge. Define W, NW, N, E, SE, S as west, north-west, north, east, south-east, and south respectively. For any shortest path $\vec{P}_i$ in $\vec{G}$ corresponding to vertices (s_i, t_i), there must exist a shortest path P_i in G that has a strictly smaller weight than $\vec{P}_i$. For this to be the case, P_i must have at least one vertex u that goes W, NW, or N to get to the next vertex on the path. This is because if edges along the path P_i only go S, SE, or E, then $\vec{P}_i$ would also be a shortest path in G since the shortest path in $\vec{G}_i$ optimizes over all paths that allow for moving S, SE, or E relative to each vertex, resulting in a contradiction.

Define a vertex $p \in P_i$ as being *disoriented* if and only if must move W, NW, or N to go from p to the next vertex on the path. Now suppose $u = (j, k)$ is the first disoriented vertex in P_i. Observe that since u is the first disoriented vertex, this implies that for the path to find its way to t_i, the path will have to either intersect column k or row j of the grid at some vertex w before the path can work its way to t_i. In either case, the cost from going directly from u to w is at least as cheap as the path taken from u to w with the first move along that path being a W, NW, or N

move. Thus, we can instead replace the round-about path from u to w with a direct path going either only S or only E. If we work along the path and apply this reasoning to each disoriented vertex we see, this implies that P_i has a path weight that is equal in weight to some path P_i' that has no disoriented vertices. Since P_i' is a shortest path and is comprised of no disoriented vertices, this implies it must also be a shortest path in $\vec{G}$ and thus $\vec{P}_i$ must have the same path cost as P_i. This is a contradiction, thus proving our claim. ■

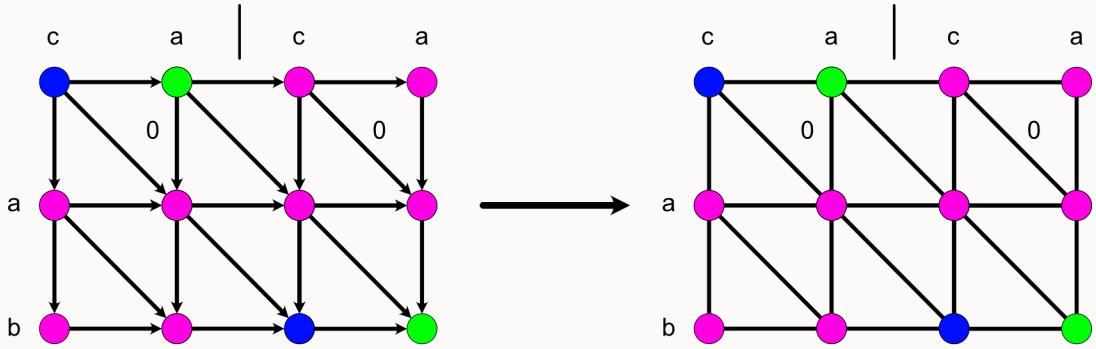

Figure 8: Convert directed planar map $\vec{G}$ to undirected planar map G

Lemma 3.3 (Cyclic Edit Distance to Edge-disjoint (s^*, t^*)-paths) *We can reduce the cyclic edit distance problem instance (s_1, s_2) to a minimum (s^*, t^*)-cut problem instance (H^*, s^*, t^*), where H^* is an unweighted, undirected planar map. The cyclic edit distance value of instance (s_1, s_2) is equivalent to the minimum (s^*, t^*)-cut value of the instance (H^*, s^*, t^*).*

Proof. To begin, we know that our input strings s_1 and s_2 induce a directed acyclic planar map $\vec{G}$ where the cyclic edit distance value is equivalent to the shortest path weight across $O(m)$ shortest (s, t)-paths in the graph $\vec{G}$. By Lemma 3.2, we know we can convert our directed planar map $\vec{G}$ to an undirected planar map G and conserve shortest paths, as shown in Figure 8. Now suppose that we have some source $s_i = (q, r)$, define column$(s_i) = r$. By definition of our initial directed acyclic map $\vec{G}$, column$(s_j) \leqslant$ column$(s_i) \leqslant$ column(s_k) for $j \leqslant i \leqslant k$ and similarly for the sinks t_i. This structure implies that we can attach a weight 0 edge from t_i to s_i for each i and construct a new graph H where if we set the faces s^* and t^* as shown in Figure 9, H becomes the input to finding the *shortest essential cycle*. Notice that the weight 0 edges we added ensure that the shortest essential cycle weight is equal to the minimum weight (s_i, t_i)-path across all i.

From lecture material on minimum cuts in planar graphs [8], we observe that a set of edges X in H correspond to a shortest essential cycle if and only if the set of their dual edges X^* correspond to a minimum (s^*, t^*)-cut in H^*, where the weight of a dual edge is equal to

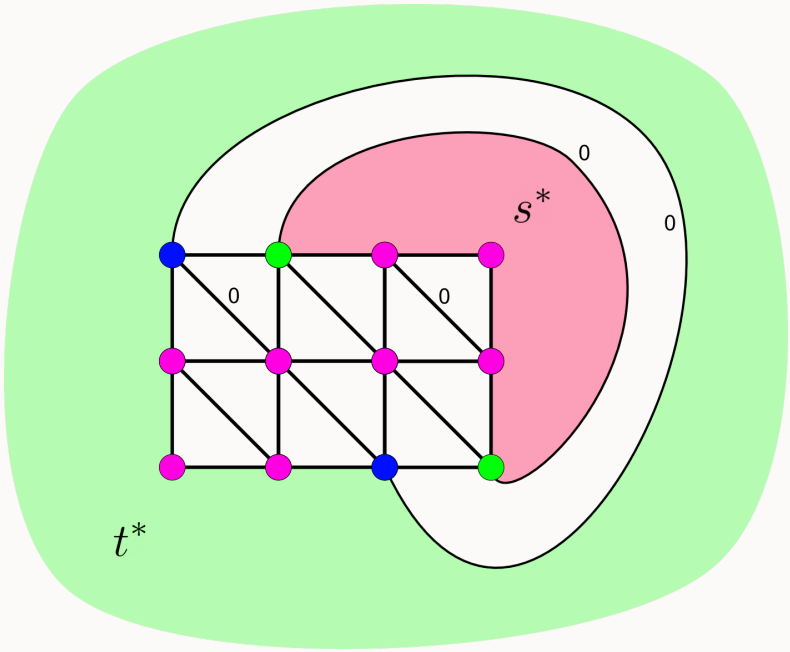

Figure 9: Converting G to H by attaching weight 0 edges between (s_i, t_i) pairs and setting inner and outer faces s^* and t^* respectively

the weight of its corresponding primal edge. Thus we can find the shortest essential cycle weight, and equivalently the cyclic edit distance value, by forming the dual map H^* like in and computing the weight of its minimum (s^*, t^*)-cut. Since the weights for our problem are either 0 or 1, we can delete the weight 0 edges and compute a minimum (s^*, t^*)-cut on H^* as an unweighted graph, this map shown in Figure 10. By the edge-disjoint version of Menger's Theorem [15], the minimum (s^*, t^*)-cut of H^* is equivalent to the maximum number of edge-disjoint (s^*, t^*)-paths in H^*. Thus, the cyclic edit distance of our strings s_1 and s_2 is equivalent to finding the maximum number of edge-disjoint (s^*, t^*)-paths in H^*, an example shown in Figure 11. ■

Lemma 3.3 tells us that if we want to compute the cyclic edit distance for our input strings s_1 and s_2, we can instead form an unweighted, undirected dual map H^* along with some special vertices s^* and t^* and feed (H^*, s^*, t^*) as an input into an edge-disjoint (s, t)-path algorithm. For us to hit our target runtime, we need to use an especially efficient algorithm for this problem. It turns out that such an algorithm was described by Karsten Weihe in 1997, the key result described in Theorem 3.4.

Theorem 3.4 (Edge-disjoint (s, t)-paths in Undirected Planar Graphs [26]) *Given an undirected, unweighted planar map G with n vertices and two vertices $s, t \in V(G)$, the number of edge-disjoint (s, t)-paths can be computed in $O(n)$ time and $O(n \log(n))$ bits of space.*

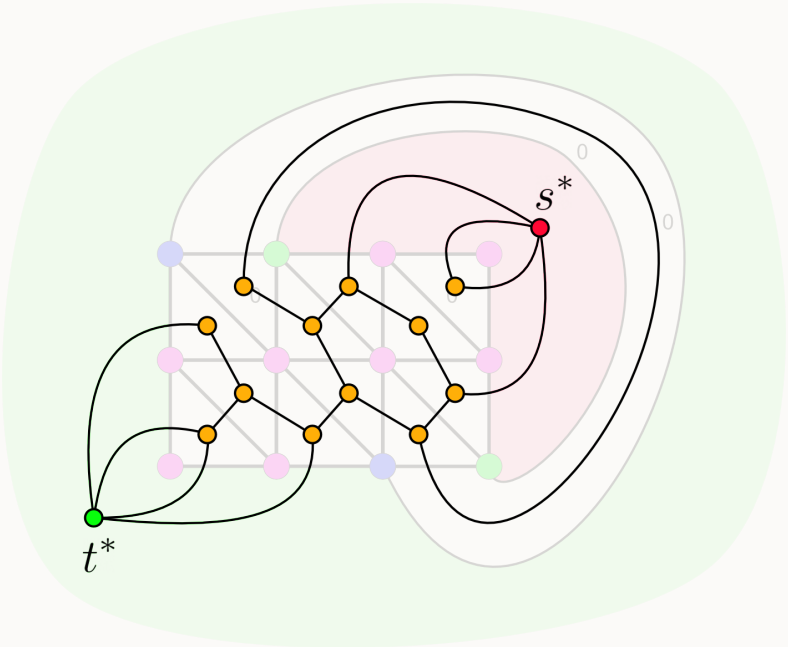

Figure 10: Form dual map H^* from H and delete all weight 0 edges

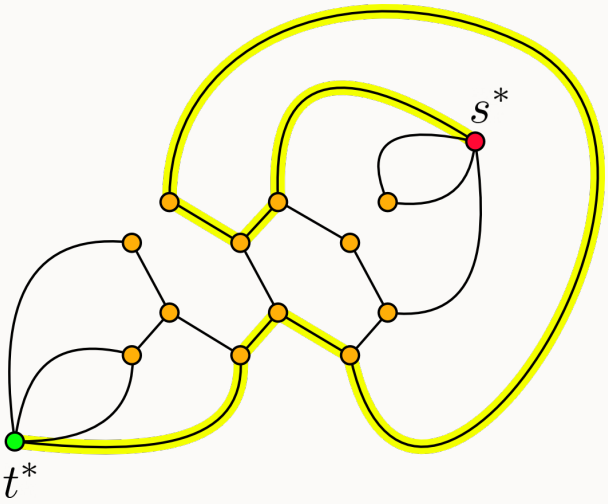

Figure 11: An example of the maximum number of edge-disjoint (s^*, t^*)-paths in H^*

We can use Weihe's edge-disjoint (s, t)-path algorithm as a subroutine in computing the cyclic edit distance, giving us a way described more formally by Algorithm 3 to compute the maximum number of edge-disjoint paths of input (H^*, s^*, t^*) in $O(mn)$ time. To see this, note first that the most faces in the undirected map H are triangular faces. Since there are $O(mn)$ triangular faces in G and each face corresponds to a vertex in the dual map, this implies that the number of vertices in H^* is $O(mn)$. By Theorem 3.4 computing the maximum number of edge-disjoint (s^*, t^*)-paths can be done in $O(mn)$ time. Thus, all we need to do is construct the reduction from the input strings (s_1, s_2) to the desired dual map configuration (H^*, s^*, t^*) in $O(mn)$ time. Since majority of the faces in H are triangles and are readily determined by the grid embedding of the planar map, we can directly form (H^*, s^*, t^*) from the input strings (s_1, s_2) in $O(mn)$ time. Since both H^* and Weihe's subroutine require at most $O(mn \log(n))$ bits of space, this implies we have an $O(mn)$ time algorithm that uses at most $O(mn \log(n))$ bits of space. Thus, we have proven Theorem 3.1.

Algorithm 3 Cyclic Edit Distance Algorithm using Planar Map Duality

```
1: procedure DUAL-CYCLIC-EDIT-DISTANCE(s_1, s_2)
2:     Use (s_1, s_2) to build dual map H* and set vertices s*, t*
3:     return WEIHE-EDGE-DISJOINT-PATHS(H*, s*, t*)
4: end procedure
```

4 Experimental Results

Within this section, we go over some experimental results to help contrast the practical use of the algorithms discussed in this report. The software was written in C++ and run on a 2019 MacBook Pro with an 8-core 2.3 GHz Intel Core i9 processor and 16 GB 2667 MHz DDR4 RAM. All implementations were written to be run on a single core. The experiments included constructing random input strings s_1 and s_2, both of equal size n for different values of n, where the random strings were comprised of ASCII characters. Experiments were performed for random string sizes from the set $\{10, 20, 100, 250, 500, 750, 1000\}$. For two randomly constructed strings s_1 and s_2 of length n, each algorithm was run 10 times with these inputs and had their runtimes averaged to help remove some variance from the time estimates. The results are shown in Figure 12 on a log-log plot to help us get a sense of the growth rate for some of these algorithms.

If we investigate the results in Figure 12, there is a handful of observations we can make. For $n = 1000$, the order of algorithms from fastest to slowest is the Kim-Park algorithm, the

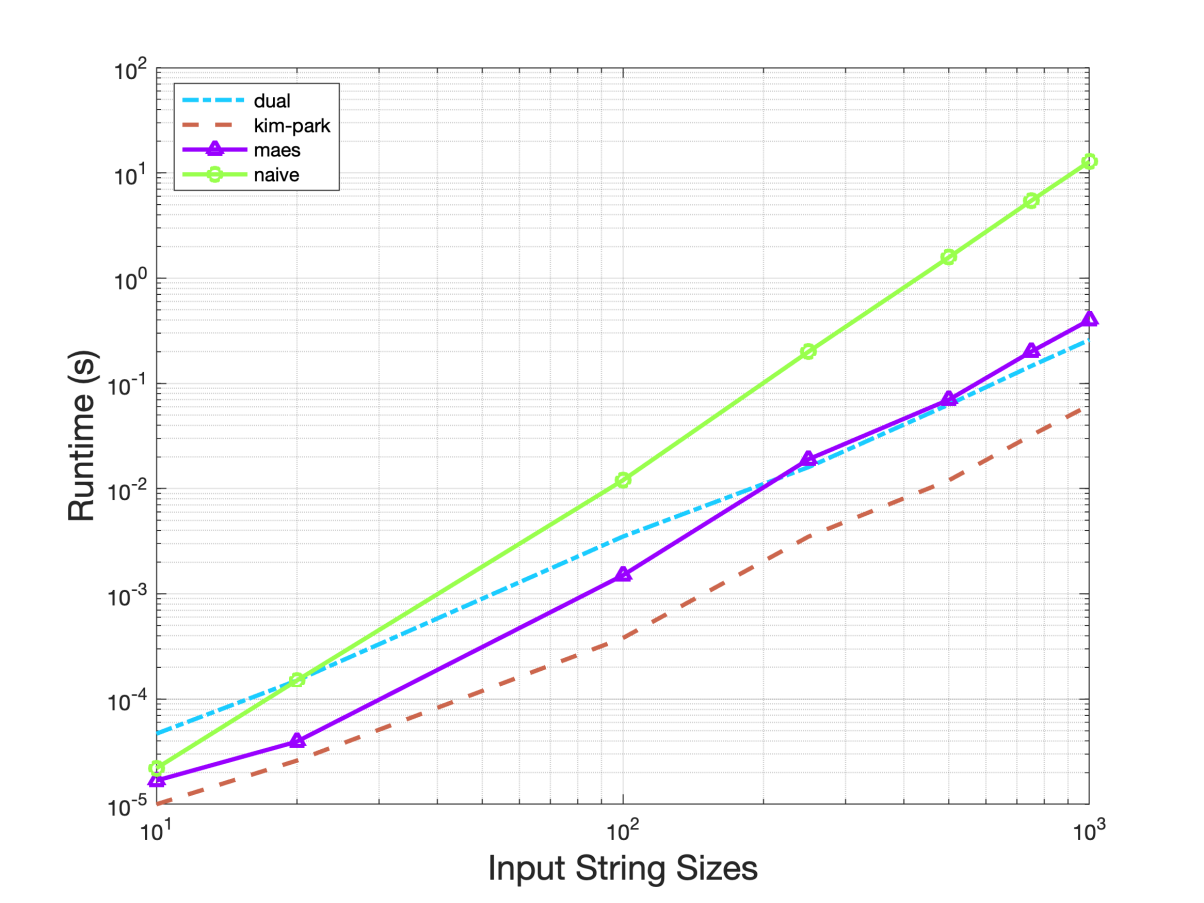

Figure 12: Experiment results between the naive algorithm, Maes' algorithm, the Kim-Park algorithm, and the dual planar map algorithm.

duality-based algorithm, Maes' algorithm, and the naive algorithm. The Kim-Park algorithm appears to have a runtime gap that I believe can be explained largely by the fact the boundary sweep algorithm used for a single update of Kim and Park's data structure ensures that most of the $O(mn)$ elements in the difference representation of the D-table are never touched, which must lead to a really small constant factor for the $O(mn)$ algorithm using this data structure. It also helps that Kim and Park's data structure uses a small amount of space, likely helping with cache performance that further drops the runtime constants of the implementation.

A separate that that is interesting to see is that with the implementations used in this experiment, Maes' algorithm has a runtime that tracks quite closely to the duality-based algorithm, while the quadratic time algorithm using Kim and Park's data structure produces a notable runtime gap once inputs gets sufficiently large. It might be worth pointing out as well that for string inputs of size less than 20, the duality-based algorithm is actually the slowest among all of them implying there are some hidden constants in the implementation that are a bit costly for small inputs. One last comment is that if one very coarsely measures the general slope of the various algorithms, we find by the fact this is a log-log plot that indeed the naive algorithm runtime grows proportional to roughly n^3, while the other methods have a runtime that grows at a rate proportional to n^2. At least to some degree, the implementations mirror some of the theoretical guarantees on the runtime.

5 Conclusion and Future Work

In this report, we covered a variety of algorithms for tackling the cyclic edit distance problem, some previously known and a new $O(mn)$ time one based on ideas from duality of planar maps. We learned a variety of techniques for how to make the cyclic edit distance problem feasible, ranging from divide and conquer strategies, to dynamic edit distance data structures, to using a variety of reductions in the planar map setting to convert the cyclic edit distance problem into what some might consider a distantly related problem of computing the maximum number of edge-disjoint paths in a planar graph. We saw through experiments that the algorithm based on Kim and Park's work performed well while the duality-based algorithm came in second as the inputs got large enough. We also saw that while Maes' algorithm is an $O(mn \log(m))$ algorithm theoretically, it ran similarly to a quadratic time algorithm for the input sizes we chose.

To take this work further, there are some minor ideas I have for improving the constant in front of the duality-based algorithm, ultimately tied to details in Weihe's edge-disjoint paths algorithm. From a theoretical perspective, I also think there could be some benefit to seeing how an approximation algorithm might materialize for the cyclic edit distance problem when

we view it through the lense of planar maps and duality. One place I investigated but did not write about in this report was that of *approximate Distance Oracles*. In some of the literature I skimmed on the idea [3–5, 10, 13, 22], there were quite a few $(1 + \epsilon)$-approximations in the planar graph setting but unfortunately many of these methods require preprocessing time to construct data structures that is ultimately a lot slower and expensive memory-wise than most of the exact methods discussed in this report. The planar graphs given as inputs to the algorithms discussed did not have the amount of known structure as what we get from the cyclic edit distance DAG, so it seems reasonable there could be a way to improve some of these results and turn them into a feasible algorithm for the cyclic edit distance problem. A lot of the algorithms benefit from r-divisions which seem to be trivial to compute efficiently for our planar map due to the grid structure, but the bottle necks in the algorithms seem to arise when needing to perform shortest path computations with the dense distance graph constructed by the r-divisions, which are less efficient than what can be done using the edit distance recursion or a breadth-first search. Still, I want to think there might be a way to get around this and so this could be interesting to investigate further.

References

[1] Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239.

[2] Bunke, H. and Bühler, U. (1993). Applications of approximate string matching to 2d shape recognition. *Pattern recognition*, 26(12):1797–1812.

[3] Chan, T. M. and Skrepetos, D. (2019). Faster approximate diameter and distance oracles in planar graphs. *Algorithmica*, 81(8):3075–3098.

[4] Charalampopoulos, P., Gawrychowski, P., Mozes, S., and Weimann, O. (2019). Almost optimal distance oracles for planar graphs. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 138–151.

[5] Cohen-Addad, V., Dahlgaard, S., and Wulff-Nilsen, C. (2017). Fast and compact exact distance oracle for planar graphs. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 962–973. IEEE.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[7] Erickson, J. (1999). Algorithms.

[8] Erickson, J. (2020). Minimum cuts. http://jeffe.cs.illinois.edu/teaching/topology20/notes/16-minimum-cut.html. Accessed: 2020-12-03.

[9] Jiang, T., Lin, G., Ma, B., and Zhang, K. (2002). A general edit distance between rna structures. *Journal of computational biology*, 9(2):371–388.

[10] Kawarabayashi, K.-i., Klein, P. N., and Sommer, C. (2011). Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 135–146. Springer.

[11] Kim, S.-R. and Park, K. (2004). A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312.

[12] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[13] Madkour, A., Aref, W. G., Rehman, F. U., Rahman, M. A., and Basalamah, S. (2017). A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*.

[14] Maes, M. (1990). On a cyclic string-to-string correction problem. *Information processing letters*, 35(2):73–78.

[15] Menger, K. (1927). Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115.

[16] Mosig, A., Hofacker, I. L., and Stadler, P. F. (2006). Comparative analysis of cyclic sequences: viroids and other small circular rnas. In *German Conference on Bioinformatics*. Gesellschaft für Informatik eV.

[17] Neuhaus, M. and Bunke, H. (2006). Edit distance-based kernel functions for structural pattern classification. *Pattern Recognition*, 39(10):1852–1863.

[18] PalazóN-GonzáLez, V. and Marzal, A. (2012). On the dynamic time warping of cyclic sequences for shape retrieval. *Image and Vision Computing*, 30(12):978–990.

[19] Palazón-González, V., Marzal, A., and Vilar, J. M. (2014). On hidden markov models and cyclic strings for shape recognition. *Pattern Recognition*, 47(7):2490–2504.

[20] Sankoff, D. (1972). Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6.

[21] Sidorov, G., Gómez-Adorno, H., Markov, I., Pinto, D., and Loya, N. (2015). Computing text similarity using tree edit distance. In *2015 Annual Conference of the North American Fuzzy Information Processing Society*

(NAFIPS) held jointly with 2015 5th World Conference on Soft Computing (WConSC), pages 1–4. IEEE.

[22] Thorup, M. (2004). Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM (JACM)*, 51(6):993–1024.

[23] Velichko, V. and Zagoruyko, N. (1970). Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, 2(3):223–234.

[24] Vintsyuk, T. K. (1968). Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57.

[25] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173.

[26] Weihe, K. (1997). Edge-disjoint (s, t)-paths in undirected planar graphs in linear time. *Journal of Algorithms*, 23(1):121–138.