Anatomy of the Cryptographic Bot DelineCrypto

Richard Pham

July 2023

1 Overview

The field of applied cryptography has been greatly enabled by computer programming languages in both the commercial and scientific spheres of development.[Cha] Modern-day computing power is multitudes beyong what it was a mere two decades ago, and this fact alongside automation techniques help diversify and strengthen cryptographic systems. Some big terms in the science of cryptography are cryptographic hash algorithms, substitution-permutation networks, Rivest-Shamir-Adleman, and symmetric/asymmetric key schemes, some of which may overlap in concept with each other.[Nis]

DelineCrypto is a bot that encrypts each input character into an integer. Given a stream of characters, it produces a concatenation of such integers. From a high-level perspective, there are several components in this bot system that enable a user (but does not guarantee) to produce cryptographically secure ciphertext. The bulk of this paper is dedicated to explaining the anatomy of the actual *DelineCrypto* program and does not go into the various proposals to implement the system for a network of users.

The first component of DelineCrypto is the delineation process that, in turn, requires a labelled dataset of float values (null values are not permitted). There are several delineating modes for the delineation process, and are to produce geometric structures in Euclidean space that classify the points of the dataset that the delineating process occurs on. In addition to the different delineating modes that delineation can occur on, its end product as a geometric structure that "encircles" all wanted points in a space has two classification schemes, hierarchical and base (child-most).

DelineCrypto's second component is a "traveller" that modifies or traverses the delineation structure using pointer references. This traveller acts on commands given to it by two files of different classes called *DInst* and *Rinst* to produce numerical sequences. The activity conducted by this "traveller" underpins an important aspect of secure communications termed *security through real-time communications*. Sections 6 and 7 exemplifies this concept in definitional detail.

The third component is a permutation graph, the data structure featured in the NP-Complete problem, "token-swapping". This structure is the cipher

of *DelineCrypto*. The vertex identifiers denote the integer value, and the token identifies the character. There are three classes of permutation commands that can be given to the data structure that will be elaborated in Section 3 on the permutation graph.

DelineCrypto's fourth component comprises of two files, of two different classes, that can be manually written or generated. The first class is the *DInst* file, that encapsulates the command sequences for each character of the character set. The file must have exactly as many command sequences as the length of the alphabet. The second class is the *RInst* file. This file "reacts" on the *DInst* file every time an output sequence of the traveller satisfies some condition stated in the *RInst* file. These two files constitute the instructions for the security agent "traveller". The "traveller" produces *private ciphertext* for the message it encrypts.

The fifth component is the bot (called *DCBot*) that when given inputs including two files, one of type *RInst* and the other *DInst*, encrypts each input character that is passed through it. A message that is to be encrypted then, is passed through this bot, and the bot outputs a sequence of values for the message's stream of characters. The sixth component is the corresponding bot that decrypts an encrypted message. Section 5 provides a summarization of *DCBot* based on the components detailed in the previous sections.

Section 7 discusses possible routes of attack on communications using *DelineCrypto*, but falls short of providing thorough elaborations on sophisticated numerical methods inevitably deployed by advanced adversaries against cryptosystems.

In the ending section of this paper, some notions are expressed on how this project came into fruition and includes other extraneous non-technical details. This paper makes the assumption that the reader has a preliminary understanding of computing terms such as "ciphertext" and is fluent in mathematical language that is used to quantify on the technical aspects of *DelineCrypto*, a cryptographic program that uses secret-key ciphers.

2 Delineation

This section provides a description of the delineation process of an n-dimensional dataset. Delineation produces a geometric solution in Euclidean space that does not use additional mappings to fit points to their wanted labels, and can be used as a stepping stone towards supervised machine-learning.

The first part on two-dimensional delineation is used to build up to n-dimensional delineation.

2.1 Two-Dimensional Delineation

Formal definitions that directly pertain to this bot's process of delineation are in order. These definitions rely on mathematical principles such as sorting and Euclidean space arithmetic.

Definition 2.1. Two-Dimensional Delineation. A binary classifier that when given a three-dimensional dataset D, with two columns representing the value of the point in two-dimensional Euclidean space and the other column representing the corresponding labels for those points, and a target label L, acts upon D by a clockwise or counter- clockwise manner and produces a **square-like geometric figure consisting of four edges, each is a fitted sequence of points.** The process constructs each of these edges using a computational method X that determines the "outermost" points for that edge. The objective in calculating these four edges is to encapsulate all the points of the target label and minimize encapsulation of the number of points not of the target label to produce a geometric structure

These are the three possible methodologies used by *DelineCrypto* to construct the edges for a three-dimensional dataset D with target label L. The first two columns constitute the two-dimensional points, and the third column have those points' labels. The methodologies are *nojag*, *nodup*, and *nocross*. The process for each of these methodologies is split into two steps:

1. A carve step, taken by all of these methodologies, that collects an ordered sequence of relevant points $R \subseteq D$ along the 0 or 1 axis.

2. A refine step that reduces the size of R by removing points of non-interest according to the specific methodology.

All of these methods are deterministic, so that given the sequence of relevant points $R \subseteq D$, the two-dimensional dataset, and refinement methodology $m \in \{nojag, nodup, nocross\}$, m acted on R produces the same sequence of points R' in every iteration. The mentioned *carve* step can now be algorithmically defined by the pseudocode of the two functions, I_{next} (index of the next point in the sequence) and $CARVE$.

---

function $score_{del}(p_1,p_2,i_1,i_d)$
  $dx = p_2[i_1] - p_1[i_1]$
  if $dx \times i_d < 0$ **then return** ∞
  else
    return dx
  end if
end function

```
    function I_next(D,p,dir,i)
2:                                  ▷ STEP: collect all the points of D that score best
                                    ▷ get axis of interest, increasing or decreasing
4:      if dir ∈ {l, r} then
            i_a ← 1
6:      else
            i_a ← 0
8:      end if
        best_score ← score_del(p, D.row(i), i_a, 1)
10:     D_2 ← {D.row(i)}
        i_2 ← i + 1
12:     while i_2 < |D.rows| do
            score_2 ← score_del(p, D.row(i_2), i_a, 1)
14:         if best_score = score_2 then
                D_2.append(D.row(i_2))
16:         else
                break
18:         end if
            i_2 = i_2 + 1
20:     end while
                                                    ▷ do tie-breaker
22:     i_a ← (i_a + 1)  mod 2
        D_2.sort(axis = i_a)
24:     if dir ∈ {l, b} then
            D_2.reverse()
26:     end if

28:     D[i : i_2] ← D_2
        D.delete_row(0)
30:     return D_2[0]
    end function
```

```
function CARVE(D,dir)
    ▷ D is an sequence of two-dimensional points ordered by the 0-axis if dir ∈ {t, b}, otherwise by the 1-axis.
    if dir ∈ {l, r} then
        i_a ← 1
    else
        i_a ← 0
    end if
    D.sort(axis = i_a)
    i ← 1
    p ← D[0, :]
    D_2 ← {p}
    while i < |D.rows| do
        p ← I_next(D, p, dir, 1)
        D_2.append(p)
    end while
    return D_2
end function
```

The iterative bounds for carving out each edge of a dataset ordered by the axis of interest with number of elements n is $O(n)$, clearly demonstrated by the pseudocode's singular direction of travel across the ordered sequence of points. Before moving on to the main function for delineation, *DELINEATE*, there are two aspects that need to be algorithmically described.

1. Refinement methodologies.
2. Fitting a sequence of two-dimensional points ordered by an axis.

2.1.1 Refinement Methodologies

There are three methodologies in use by this bot that refine a point sequence representing one of the four edges of a dataset D with target label L. This section uses the example found in Figure 1 to exemplify these methodologies. The initial carve of the four edges L,R,T,B are illustrated in Figures 2- 5, respectively.

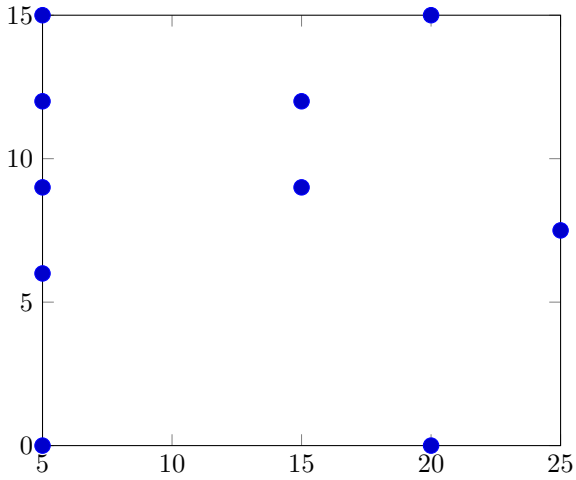

Figure 1: Example two-dimensional dataset.

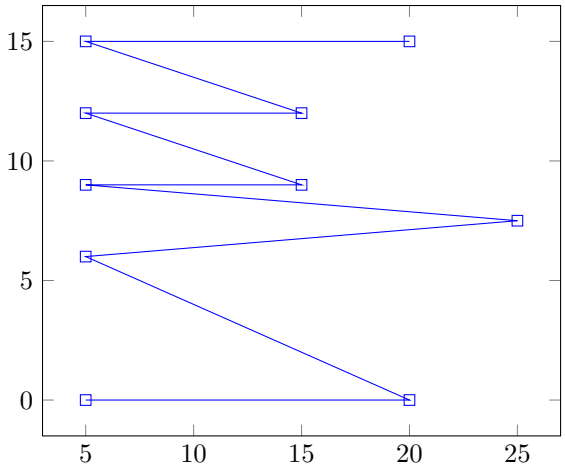

Figure 2: Left carve of Figure 1 dataset.

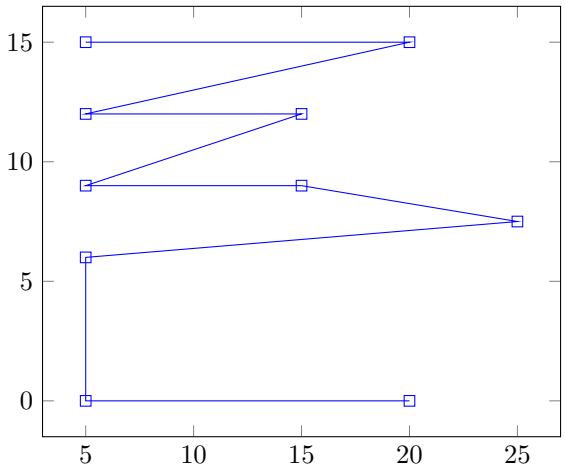

Figure 3: Right carve of Figure 1 dataset.

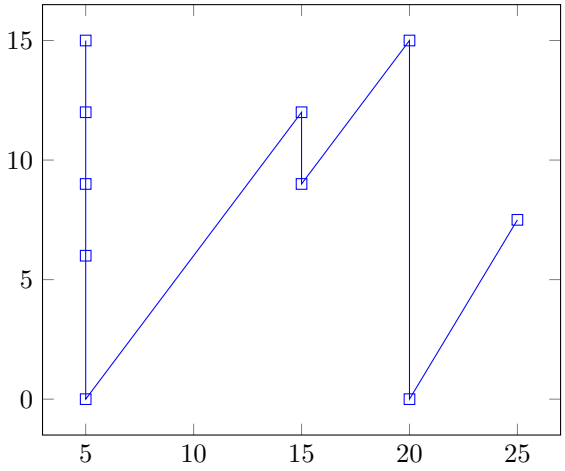

Figure 4: Top carve of Figure 1 dataset.

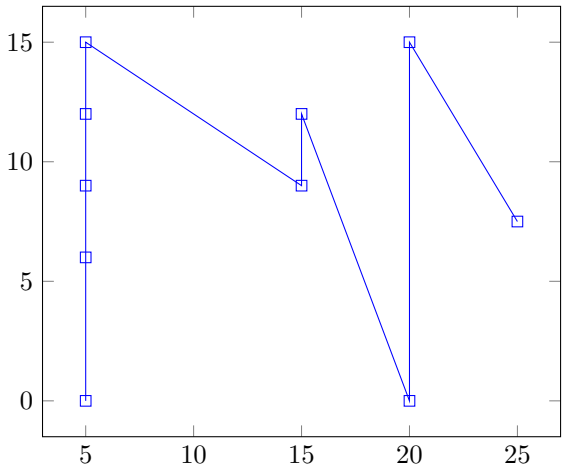

Figure 5: Bottom carve of Figure 1 dataset.

In the section 2.1.3, the classification method of a delineation structure is described, and these illustrations demonstrate the error-prone nature of carving out edges for a delineation without refining them. The paper now proceeds to focusing on the refinement methodologies.

The first one presented is the *nojag* function, which takes as input a point sequence representing an edge and a direction d. The *nojag* refinement attempts to remove all points of the point sequence E that fit the programmed definition of a "jag".

Definition 2.2. Jag. Given a direction $d \in \{l, r, t, b\}$ and pair of two- dimensional points p_1 and p_2, a jag occurs between p_1 and p_2 if for the axis of relevance a (0 if $d \in \{l, r\}$, otherwise 1), $c = p_2[a] - p_1[a]$ is positive for $d \in \{l, b\}$ or negative for $d \in \{r, t\}$.

The *nojag* function can be described by the pseudocode below.

```
    function NOJAG(E,dir)
2:      i = 0
        d = {}
4:      while i < |E| − 1 do
            if IS_JAG(E.row(i), E.row(i + 1), dir) then
6:              d.append(i + 1)
            end if
8:          i ← i + 1
        end while
10:     delete_rows(E, d)
        return E
12: end function
```

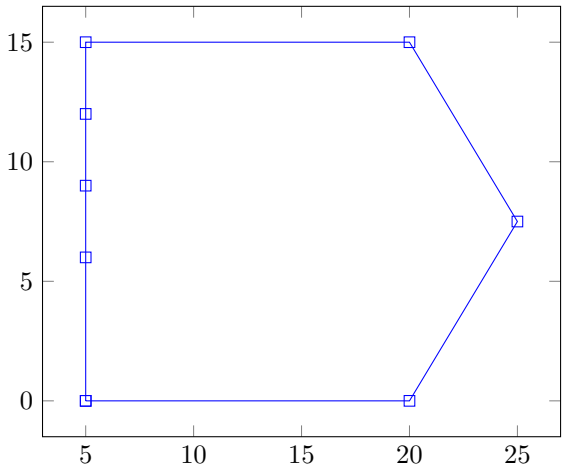

Figure 6: Refined edges of Figure 1 dataset by *nojag*

The *nocross* function takes as input an axis and a pair of point sequences representing complementary edges, i.e. left,right or top,bottom. It refines the pair of point sequences by removing points from them that result in something termed a rectangle cross.

Definition 2.3. Region-Point Intersection. In the two-dimensional space, given two pairs of points e_0 and e_1 that each constitute a line, and a point p, p intersects the region of e_0 and e_1 if the following two conditions are true:

$$p[0] \geq min(e_0[0], e_1[0]) \;\; \& \;\; p[0] \leq max(e_0[0], e_1[0]),$$

$$p[1] \geq min(e_0[1], e_1[1]) \;\; \& \;\; p[1] \leq max(e_0[1], e_1[1]).$$

Definition 2.4. Rectangle cross. Given two pairs of points e_0 and e_1 that each constitute a line, and those lines constitute a region A as described in Definition 2.3, a rectangle cross occurs when one of the points $e_0[0]$ or $e_1[0]$ and A have a *region-point intersection.*

Definition 2.4 serves as a basis for a thorough pseudocode description of the *nocross* refinement methodology. There are two aspects to *nocross* refinement, identification of the points in region-point intersections and a corrective method for such points. The method *CORRECT_RC* decides on the removal of each of the start points of two line segments e_0 and e_1 by using another method *CLOSER_SEGMENT* that determines which line segment,one of e_0 or e_1, that $e_x[0]$ is closer to.

```
function CLOSER_SEGMENT(p,e_0,e_1,axis)
    d_0 ← |e_0[0, axis] − p[axis]|
    d_1 ← |e_1[0, axis] − p[axis]|
    if d_0 < d_1 then
        return 0
    else
        return 1
    end if
end function
```

```
function CORRECT_RC(e_0,e_1,axis)
    c_0 ← is_region_point_intersection(e_0, e_1, e_0[0])
    c_1 ← is_region_point_intersection(e_0, e_1, e_1[0])
    b_0 ← true
    b_1 ← true
    if c_0 then
        if CLOSER_SEGMENT(e_0[0], e_0, e_1, axis) = 1 then
            b_0 ← false
        end if
    end if
    if c_1 then
        if CLOSER_SEGMENT(e_1[0], e_0, e_1, axis) = 0 then
            b_1 ← false
        end if
    end if
    return {b_0, b_1}
end function
```

A last method that *nocross* requires is one that determines the next point of interest in the second point sequence e_1 of the arguments given.

```
function NEXT_POINT_INDEX(E,p_0,p_1,axis,start_index)
    r_0 ← {p_0[axis], p_1[axis]}
    r ← {min(r_0), max(r_0)}
    i ← start_index
    while i < |E.rows| do
        if E[i, axis] ≥ r[0]  &  E[i, axis] ≤ r[1] then
            return i
        end if
        i ← i + 1
    end while
    return −1
end function
```

```
    function NOCROSS(E_0,E_1,axis)
2:                                  ▷ collect all the points of E_0 and E_1 that do not
                                    ▷ result in rectangle crosses.
4:      v_0 ← {0}
        v_1 ← {0}
6:      E_10 = {E_0.row(v_0[−1])}
        E_11 = {E_1.row(v_1[−1])}
8:      i_0, i_1 ← 1, 1
        axis_2 ← (axis + 1)  mod 2
10:     while i_0 < |E_0.row|  &  i_1 < |E.1.row| do
            e_0 ← {E_0.row(v_0[−1]), E_0.row(i_0)}
12:         i_11 ← NEXT_POINT_INDEX(E_1, e_0[0], e_0[1], axis, i_1)
                                    ▷ CASE: no relevant points, increment i_0
14:         if i_11 = −1 then
                v_0.append(i_0)
16:             i_0 ← i_0 + 1
                E_10 = {E_0.row(v_0[−1])}
18:             continue
            end if
20:                                                                  ▷
            while i_1 < i_11 do
22:             v_1.append(i_1)
                i_1 ← i_1 + 1
24:             E_11 = {E_1.row(v_1[−1])}
            end while
26:         e_1 ← {E_1.row(v_1[−1]), E_1.row(i_1)}
            b_0, b_1 = CORRECT_RC(e_0, e_1, axis_2
28:         if b_0 then
                v_0.append(i_0)
30:             E_10 = {E_0.row(v_0[−1])}
            end if
32:         if b_1 then
                v_1.append(i_1)
34:             E_11 = {E_1.row(v_1[−1])}
            end if
36:         i_0 ← i_0 + 1
            i_1 ← i_1 + 1
38:     end while
        return E_10, E_11
40:
    end function
```

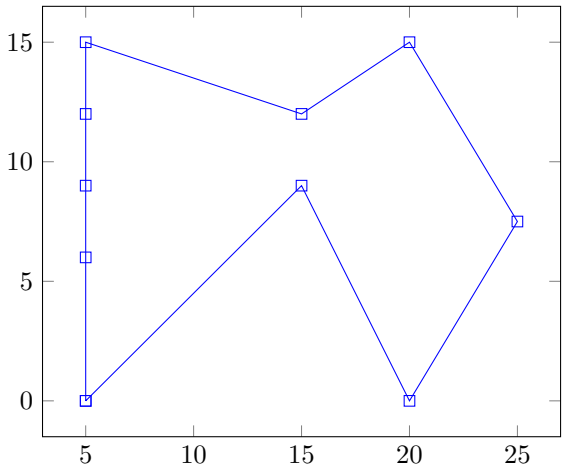

Figure 7: Refined edges of Figure 1 dataset by *nocross*

The iterative runtime for *nocross*, which to clarify, takes as arguments two complementary point sequences ordered by the axis argument, is $O(n \times log(n))$, where n is the value $max(\{|E_0.rows|, |E_1.rows|\})$. The factor n of this runtime is attributed to the first "while" loop (see line 10) and the other factor $log(n)$ is attributed to the iteration in that "while" loop that calls *NEXT_POINT_INDEX* on the point sequence E_1.

The last methodology that *DelineCrypto* has at its disposal is the *nodup* function. In short, *nodup* takes as input the same arguments as that for *nocross*, and for the point sequences E_0 and E_1, determines for each point p found in both E_0 and E_1 which sequence (E_0 or E_1) to delete p from, while the other sequence gets to keep p, now a non-duplicate point. The decision for choosing which point sequence to delete any duplicate point p from is described in the function *ASSIGN_DUPLICATE_POINT*.

```
    function ASSIGN_DUPLICATE_POINT(E0,E1,p,axis)
2:      mean0 ← mean(E0)
        mean1 ← mean(E1)
4:      d0 ← (|mean0[axis] − p[axis]|)/|E0.rows|
        d1 ← (|mean1[axis] − p[axis]|)/|E1.rows|
6:      if d0 < d1 then
            delete_point(E1, p)
8:      else
            delete_point(E0, p)
10:     end if
    end function
```

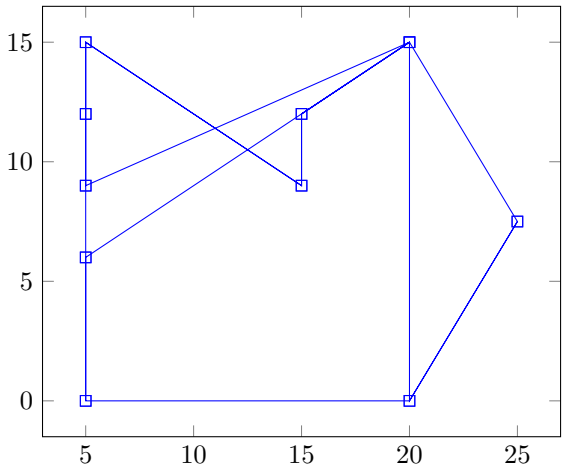

Figure 8: Refined edges of Figure 1 dataset by *nodup*

The implementation of *DelineCrypto*'s *nodup* refinement has an iterative runtime of $O(N^2)$. In Figure 8, the edges of the delineation are difficult to visually differentiate due to how the point-elimination of *nodup* method works so to compare, refer to Figures 2-4 for the original edges.

2.1.2 Fitting a Point Sequence

Definition 2.5. Two-Dimensional Point Sequence Fit. A *point sequence* sorted by one of its two axes and pertains to one of the four edges in a two-dimensional delineation is fitted by applying a *pairwise point-fit* between every pair of consecutive points in the sequence.

There are numerous approaches to "fitting" two points in two-dimensional space. To fit, in this context, means to interpolate all the intermediary points in between an arbitrary pair of points. *DelineCrypto* chooses between three schemes to fit a pair of two-dimensional points. These schemes are designated *LogFit22*, *SquareFit22*, and *LineFit22*.

Definition 2.6. LogFit22. Given a pair of two-dimensional points p_1 and p_2, interpolation by the x-value that sits between $p_1[0]$ and $p_2[0]$ is done by the function F_l accompanied by r,

$$r(x, p_1, p_2) = \frac{|x - p_1[0]|}{|p_1[0] - p_2[0]|},$$

$$F_l(x, p_1, p_2) = p_1[1] + \frac{log(r(x, p_1, p_2) \times 9 + 1)}{log(10)} \times |p_1[1] - p_2[1]|.$$

Definition 2.7. SquareFit22. Given a pair of two-dimensional points p_1 and p_2, interpolation by the x-value that sits between $p_1[0]$ and $p_2[0]$ is done by the

function F_s accompanied by r (previously defined in definition 2.3),

$$F_s(x, p_1, p_2) = p_1[1] + r(x, p_1, p_2) \times |p1[1] - p2[1]|.$$

Definition 2.8. LineFit22.

$$F_n(x, p_1, p_2) = \begin{cases} m(p_1, p_2) \times x + b(p_1, p_2) \texttt{ if slope is not undefined} \\ p_1[1] \texttt{ otherwise} \end{cases}$$

in which m is the function that determines the slope of p_1 and p_2 and b the function that determines the y-intercept of p_1 and p_2.

2.1.3 The Method *Delineation*

After describing the various methodologies used to refine and fit edge representations, there is still the matter of the decision function that determines how to fit each contiguous pair in a point sequence, out of the possibilities

$$\{LogFit22, SquareFit22, LineFit22\}.$$

The following notation is used when referring to any one of these fits on a pair of two-dimensional points.

- Any fit on a pair of points is denoted F_x, x is its identifier.
- Declaring a fit on a pair of points p_0 and p_1 is of the form

  $$F_x(p_0, p_1).$$

- The first and second points of a fit are referenced as $F_x[0]$ and $F_x[1]$, respectively.
- Converting a fit F_x to another fit-type Q is expressed as

  $$Q(F_x).$$

- Determining the y-value of an input x' in the x-range of a fit F_x is

  $$F_x.fit(x'),$$

  and the x-value given an input y' in the y-range of F_x is

  $$F_x.fit^{-1}(y').$$

```
function REFINE_EDGES(L_E,refinement)
    if refinement = nojag then
        L_E[l] ← NOJAG(L_E[l], l)
        L_E[r] ← NOJAG(L_E[r], r)
        L_E[t] ← NOJAG(L_E[t], t)
        L_E[b] ← NOJAG(L_E[b], b)
    else if refinement = nodup then
        L_E[l], L_E[r] ← NODUP(L_E[l], L_E[r], 1)
        L_E[t], L_E[b] ← NODUP(L_E[t], L_E[b], 0)
    else
        L_E[l], L_E[r] ← NOCROSS(L_E[l], L_E[r], 1)
        L_E[t], L_E[b] ← NOCROSS(L_E[t], L_E[b], 0)
    end if
end function
```

```
function PREDELINEATE(D,target_label,refinement,clockwise)
    L_E ← []
    L_E.label ← target_label
    D_2 ← filter(D, target_label)
    L_E[l] ← CARVE(D_2, l)
    L_E[r] ← CARVE(D_2, r)
    L_E[t] ← CARVE(D_2, t)
    L_E[b] ← CARVE(D_2, b)
                                          ▷ refine the edges
    REFINE_EDGES(L_E, refinement)
                          ▷ order the edges according to clockwiseness
    if clockwise then
        L_E[r] ← reverse(L_E[r])
        L_E[b] ← reverse(L_E[b])
    else
        L_E[l] ← reverse(L_E[l])
        L_E[t] ← reverse(L_E[t])
    end if
    return L_E
end function
```

The next three functions initially fit contiguous pairs of points from the *carve* method with either a *LineFit22* (if the pair of points share the same value along an axis) or a *LogFit22*.

```
function INITIAL_FIT_FOR_PAIR(p0,p1)
    if p0[0] = p1[0] then
        return LineFit22(p0, p1)
    else if p0[1] = p1[1] then
        return LineFit22(p0, p1)
    else
        return LogFit22(p0, p1)
    end if
end function
```

```
function INITIAL_FIT_FOR_EDGE(E)
    V ← {}
    i ← 0
    while i < |E.rows| − 1 do
        V.append(INITIAL_FIT_FOR_PAIR(E.row(i), E.row(i + 1))
        i = i + 1
    end while
    return V
end function
```

```
function INITIAL_FIT_FOR_DELINEATION(L_E)
    L_F = []
    L_F.label ← L_E.label
    L_F[l] ← INITIAL_FIT_FOR_EDGE(L_E[l])
    L_F[r] ← INITIAL_FIT_FOR_EDGE(L_E[r])
    L_F[t] ← INITIAL_FIT_FOR_EDGE(L_E[t])
    L_F[b] ← INITIAL_FIT_FOR_EDGE(L_E[b])
    return L_F
end function
```

Classification is a primary objective of delineation, so this task does require a look at, with the next four methods comprising the programming to this problem.

```
function MATCHING_FITS_FOR_POINT(L_F,p,axis)
    if axis = 1 then
        f_1 ← MATCHING_FIT_ON_EDGE(L_F[l], p, axis)
        f_2 ← MATCHING_FIT_ON_EDGE(L_F[r], p, axis)
    else
        f_1 ← MATCHING_FIT_ON_EDGE(L_F[t], p, axis)
        f_2 ← MATCHING_FIT_ON_EDGE(L_F[b], p, axis)
    end if
    return f_1, f_2
end function
```

```
function MATCHING_FIT_ON_EDGE(E,p,axis)
    for f_x ∈ E do
        i_x ← {f_x[0][axis], f_x[1][axis]}
        m_0, m_1 ← min(i_x), max(i_x)
        if p[axis] ≥ m_0 & p[axis] ≤ m_1 then
            return f_x
        end if
    end for
    return ∅
end function
```

```
function CLASSIFY_POINT_ON_AXIS(L_F,p,axis)
    f_0, f_1 ← MATCHING_FITS_FOR_POINT(L_F, p, axis)
    a_2 ← (axis + 1) mod 2
    if axis = 1 then
        Z ← {f_0.fit(p[a_2]), f_1.fit(p[a_2])}
    else
        Z ← {f_0.fit^-1(p[a_2]), f_1.fit^-1(p[a_2])}
    end if
    m_0, m_1 ← min(Z), max(Z)
    if p[axis] ≥ m_0 & p[axis] ≤ m_1 then
        return L_F.label
    end if
    return −1
end function
```

function $CLASSIFY\_POINT(L_F, p)$
  $l_1 \leftarrow CLASSIFY\_POINT\_ON\_AXIS(L_F, p, 1)$
  $l_0 \leftarrow CLASSIFY\_POINT\_ON\_AXIS(L_F, p, 0)$
  if $l_1 \neq -1 \mid l_0 \neq -1$ **then**
    return $L_F.label$
  end if
  return -1
end function

To classify a point p using L_F, a *Deline22* instance represented as a map to fit-sequences along an axis, the algorithm first finds two complementary fits along the axis (T and B for axis 0,L and R for axis 1). Since the number of fits per edge has an upper-bound of $O(n)$, then finding two complementary fits to classify a point takes under $2 \times O(n)$ iterations. For axis 1, classification uses a fit's *fit(x)* function and for the other axis, $fit^{-1}(y)$. Using two fits F_0 and F_1, a point p by axis a is classified as $L_F.label$ if the value of its opposing axis falls between the output values of the two fits F_0 and F_1. So the method *CLASSIFY_POINT_ON_AXIS* has an iterative runtime of $2 \times O(N)$ by its search for the matching pair of fits for the point.

Remark. A *Deline22* instance has an attribute i that is used as a index pointing to a row of the dataset it delineates on. This attribute is relevant to the "Move One" command in Section 4.

One last method required to describe the *delineation* method is a *calibration* method that switches a *LogFit22* to a *SquareFit22* on a pair of points and vice-versa based on the net score. The objective of the calibration methodology for *delineation* is to maximize the net score of the geometric structure of delineation in the context of classifying the points that it was trained on. This objective function is used to calculate the score of a point.

$$S(L_F, p) = \begin{cases} 1 \texttt{ if } CLASSIFY\_POINT(L_F, p) = p[2] \\ 0 \texttt{ otherwise.} \end{cases}$$

And the decision function to determine whether a pair of points belonging to an edge sequence is better fitted by a *LogFit22* or a *SquareFit22* is

$$ALT(f_x) = max_{f \in \{LogFit22, SquareFit22\}} \sum_{p \in D} S(f, p).$$

For the sake of brevity, this paper does not provide pseudocode on the *calibration* method that the main method *delineation* calls at its end. The general outline of the calibration method is as follows:

- Iterate through each edge comprising of a fit sequence.

  - For each fit F, if it is a *LineFit22*, continue. Otherwise, set F to $ALT(F)$.

This calibration method classifies each point for roughly four times, one time for every edge. Using the iterative runtime for classifying one point, $4 \times O(N)$, that for *calibration* has an upper bound $O(N)$ of fits for each edge. So iterating through the fits of all four edges has the runtime of

$$(4 \times O(N)) \times (4 \times O(N)) \approx O(N^2).$$

All of the relevant methodologies for the main method of *delineation* have been described in full, and the paper can now proceed to this main method's pseudocode.

---

function $DELINEATE(D, target\_label, refinement, clockwise)$
2: $L_E \leftarrow PREDELINEATE(D, target\_label, refinement, clockwise)$
$L_F \leftarrow INITIAL\_FIT\_FOR\_DELINEATION(L_E)$
4: $CALIBRATE(L_F)$
end function

In the system *DelineCrypto*, *Deline22* is the class that implements this function.

2.1.4 Positive Errors in Delineations

The *delineation* process is able to construct a geometric shape that encapsulates all points of a target label. However, there is still the issue of positive classification errors.

Definition 2.9 (Positive Classification Error.)**.** Given a delineation L_F and a sample s, L_F classifies s as $L_F.label$ but s has a different label.

A noteworthy observation is that the procedure *delineation* takes produces an "encapsulating" geometric structure around the points of interest means that the number of positive classification errors will most likely be greater than the number of negative classification errors. **A structure called *Deline22MC* takes further delineations on those misclassified points to minimize this problem.**

2.1.5 Multi-Classification in Two Dimensions

The solution that *Deline22MC* calculates for a dataset is a sequence of delineations $L_F^{(i)}$, each with its own target label. **The classification scheme for Deline22MC uses the CLASSIFY_POINT method of Deline22, but chooses only the classification/s produced by the "child-most" delineations.**

Definition 2.10 (Child-most delineation)**.** Given a sequence S_{L_F} of delineations, a delineation L_F is child-most if it does not fully contain any other delineation in S_{L_F}.

Definition 2.11 (Containment of delineation)**.** A delineation $L_F^{(1)}$ is contained in $L_F^{(2)}$ if for every fit F_x belong to $L_F^{(1)}$, the two endpoints of F_x are positively labelled by $L_F^{(2)}$.

This is a procedure by a *Deline22MC* instance that produces a geometric structure to fit the points of a three-dimensional dataset D.

1. Set the target points S to the subsequence of points in D with the least frequent label L *that has not yet been targeted given* D. Delineate on S to produce L_F.

2. Iterate through the points of D, and for any point classified as L by L_F, delete it from D. Add L_F to *TMPCACHE*, the cache of delineations with positive classification errors.

3 . Set D' as the dataset of points misclassified by L_F that resulted in positive error. Repeat steps 1 and 2 for that dataset D' (instead of D) until positive classification errors do not exist.

4. Add the original delineation L_F and its child delineations (constructed in step 3) to *CACHE*, the sequence of delineations without positive classification errors.

5. Loop back to steps 1-4.

This procedure is recursive on each delineation it constructs for a subsequence of points in the input dataset D. **Some implementations of *Deline22MC* might result in infinite recursive loops on some datasets due to the algorithm struggling to produce delineations without positive classification error.** These infinite recursive loops are quite possible on clusters of points with different labels.

To fix this problem, set a $target\_delineation\_cap$ variable set to a small number such as 5. This variable determines the number of attempts each targeted point can be attempted to be delineated.

2.1.6 Performance Variations on Refinement and Fit

There are variations in performance between the different refinement methodologies. Performance, in the context of delineation, refers to its classification accuracy of the points fitted by the algorithm. The specific difference on the performance of one refinement methodology over another for a particular dataset rests on the interpoint Euclidean distances and geometric formation of that labelled collection of points. So constructing a bounds to express the differences in refinement methodology performance cannot be easily done. But based on some experimental data (not shown), the differences can substantially vary.

One important relation between the different refinement methodologies is

$$AREA(NOJAG) \leq AREA(NOCROSS) \leq AREA(NODUP).$$

There are cases of datasets in which one refinement methodology proves vastly superior over another. An approximation of the number of points that constitute an edge in an arbitrary dataset D is based on this relation that uses two functions based on Euclidean distance.

$$max_{distance}(D) = \max_{p_1,p_2 \in D} \|p_1 - p_2\|,$$

$$max_{point}(D,p) = \max_{p' \in D} \|p - p'\|,$$

$$\texttt{(num. of edge points)} \propto \bigcup_{p \in D} [(max_{point}(D,p)/max_{distance}(D)) \geq t]; t \in [0,1].$$

2.2 Extending Two-Dimensions to Arbitrary Multi-Dimensions

A structure called *DelineMD* of *DelineCrypto* is dedicated to delineating datasets of arbitrary $m \geq 3$ dimensions. The format of these datasets must satisfy these two arbitrary conditions specific to *DelineCrypto*'s design:

- The last column of any input dataset D to *DelineMD* is designated as the label-column.
- The set of labels consist of non-negative numerical values, typically in the space $\mathbb{N}$.

A description of *DelineMD*'s delineation method is provided.

1. For a dataset D represented as an $n \times m$ matrix, user selects column c for the axis of delineation, $0 \leq c \leq m-2$ and declares an instance D_{md} of *DelineMD*.
2. Iterate through each column i in the range $[0, m-1]$, except for the axis column c and the last column $m-1$ (used for labelling).
   (a) For each column i, set
   $$D' = [D.col(c), D.col(i), D.col(m-1)],$$
   a three-dimensional dataset.
   (b) Run multi-class delineation on D' using an instance D_{mc} of *Deline22MC*. Add D_{mc} to D_{md}'s map solution H,
   $$H : n \rightarrow Deline22MC; n \in \mathbb{N}.$$

The classification method in multi-dimensional contexts uses that of *Deline22MC* and thus relies on the concept of **child-most delineations**. The description of the classification method in multiple dimensions goes as such.

1. Algorithm takes as input a multi-dimensional point p of the same length as the number of columns as the dataset that a D_{md} has delineated on.

2. Declare an empty map L of the form

   $L : n \rightarrow \{$`l| l a positive label of point p by Deline22MC on column n`$\}$.

3. Iterate through each of the *Deline22MC* instances in the cache of D_{md}, calculate its set of positive labels P and add it to L.

2.3 Concluding Remarks on Delineation

The multi-dimensional delineation procedure of the *DelineMD* structure falls short of a machine-learning algorithm. The reason is that its delineation method produces a constant output given the same input arguments and data, and does not operate to improve any objective function past the point of obtaining that constant output. In a word, the procedure is deterministic. Furthermore, there are deficits in its classification method. Recall that the classification for an arbitrary point p by a D_{md} is the map L,

$L : n \rightarrow \{$`l| l a positive label of point p by Deline22MC on column n`$\}$.

There can be more than classification for p by each dimension. And the set of labels by each dimension may not be one and the same, so the resulting labels can be contradictory hypotheses on the classification of p.

This classification map L still has its uses. One use is determining if a point is "deviant" or not.

Definition 2.12 (Deviant point by delineation)**.** Given a threshold value $t \in [0, 1]$, a point p is t-deviant in a *DelineMD* instance D_{md} if its map of labels L,

$L : n \rightarrow \{$`l| l a positive label of point p by Deline22MC on column n`$\}$,

results in

$$|\bigcap_{n \in L}(L[n])|/|\bigcup_{n \in L}(L[n])| < t.$$

The definition of a deviant point can aid in determining how well a multi-dimensional delineation performs. Suppose that after a *DelineMD* delineates on a dataset D, and goes back to classifying all of D's points, there are x points that can be identified as being t-deviant, $t \in [0, 1]$.

Another use is in the statistical/algebraic problem of **multi-collinearity**.

Definition 2.13 (Multi-collinearity)**.** For a dataset $D = [X_0, X_1, \ldots X_{m-1}]$ of m columns (each a variable), a set

$$\{X_{i_1}, \ldots, X_{i_p}\} \in \{X_0, X_1, \ldots X_{m-1}\}$$

in which $3 \leq p \leq m$ is multi-collinear if $D' = [X_{i_1}, \ldots, X_{i_p}]$ is not linearly independent.[Sha]

Definition 2.14 specifies what linear independence of variables is by antithesis.

Definition 2.14 (Linearly independent variables)**.** A dataset $D = [X_0, \ldots, X_{q-1}]$ has q linearly dependent variables if there exists a linear combination

$$c_0 x_0 + c_1 x_1 + \cdots + c_{q-1} x_{q-1} = 0$$

that satisfies all samples (rows) of D. [Sha]

The geometric shape of a *DelineMD* instance's delineation provides an alternative to the usually numerically-intensive matrix computations used by machine-learning algorithms in determining multi-collinearity.

Definition 2.15 (Delineation collinearity)**.** For a *DelineMD* instance D_{md} that delineates over a dataset $D = [X_0, X_1, \ldots, X_{m-1}]$, two columns X_{i_0} and X_{i_1} are t-collinear for a threshold value $t \in [0, 1]$ if for at least $\lceil t \times |D.rows| \rceil$ samples,

$$D_{md}.H[i_0].classify(p) = D_{md}.H[i_1].classify(p); p \in D.$$

Note that although the term "delineation collinearity" is a bit of a misnomer, due to the non-linear nature of the delineation geometries used in classifying points, its definition is roughly as effective as orthodox collinearity calculations in determining how related two variables of a dataset are.

Another use is **dimensionality reduction**. A process that can reduce the m dimensions of a dataset D into a dataset with less variables is explained in Definition 2.16. Whereas orthodox machine-learning approaches use widely-known techniques such as *principle component analysis* [Ull, p. 8], the *DelineMD* structure takes a different approach.

Definition 2.16 (Dimensionality reduction by *DelineMD*)**.** For a *DelineMD* instance D_{md} that has delineated over a dataset D, suppose there is a decision function, D_f,

$$D_f(p, i) = \begin{cases} 1 \texttt{ if } D_{md}.H[i].classify(p) = \{p[-1]\} \\ 0 \texttt{ otherwise.} \end{cases}$$

Suppose for the i'th column,

$$\Sigma_{p \in D} D_f(p, i) < t,$$

such that $t \in [0, 1]$ is user-specified, then the i'th variable is not accurate above threshold t and should be reduced (removed) from dataset D.

DelineCrypto possesses the class *DelineMD* that sets a non-label dimension as the axis, and then uses instances of *Deline22MC* to delineate over all the other non-label dimensions. Given a dataset D with number of rows N and L labels, suppose there there are $K = \frac{N}{L}$ elements for each label. Then the iterative runtime for multi-classification is approximately

$$G(D) = O(K^2) \times (L - 1)^2,$$

given the condition that each point is targeted only once.

3 The NP-Complete Token-Swapping Problem

A permutation graph is a very important aspect in the encoding process of a character. The secrecy of *DelineCrypto*'s encryption benefits from the NP-Completeness of this structure. The reason why this NP-Complete problem was selected to be a part of this cryptographic bot *DelineCrypto* rests on the assumption that if it is difficult to find the best solution to an NP-Complete problem, then the comparably difficulty of finding a particular solution.

3.1 Defining the Token-Swapping Problem

A permutation graph is a subclass of a simple undirected graph.

Definition 3.1 (Simple undirected graph)**.** A structure G with a set of vertices V, each with its own identifier, and a set of edges E. Each edge in E is composed of at least two values, and they are its two end-vertices $u, v \in V$. Edges can include additional values such as "distance","weight",etc.

The notation for G has the following notation:

1. The set of vertices belong to G is $G.V$.
2. The set of edges belong to G is $G.E$.

In this paper, take the term "graph" to mean a "simple undirected graph" unless otherwise stated. Permutation graphs are a subclass of graphs that have two additional properties.

Definition 3.2 (Permutation graph)**.** A simple undirected graph G with the additional properties:

1. Every vertex $v \in G.V$ has an object called a *token*. Any token of $v \in G.V$ has an identifier i equal to that of some vertex $u \in G.V$. For instances that require the identifier of a token t, its identifier is the subscript, i.e. t_{id}.
2. The token placement P of G is a mapping

$$P(t_u) = v; u, v \in G.V.$$

   P is a bijective mapping from token to its vertex location. A token placement for a simple undirected graph G is $G.P$.

2. An action called a *swap* can be taken on an edge $e \in G.E$. A swap on edge e results in the two tokens t_i and t_j sitting on its vertices v_x and v_y, respectively, to then sit on vertices v_y and v_x after the swap. Swaps are denoted by s_e, s is a constant and e is the target edge that the swap occurs on. The notation for a swap s_e conducted on G is denoted $s_e(G)$.

A permutation graph is the object of focus in the NP-Complete problem, *token-swapping problem*.

Definition 3.3 (Token swapping problem)**.** Given a permutation graph G and a token placement $P' \neq G.P$, for a positive integer k, does there exist a sequence of swaps $\overrightarrow{S}$ with $|\overrightarrow{S}| \leq k$ that after conducted on G results in $G.P = P'$?

According to contemporary literature and the efforts taken on this project, there is no formulaic methodology outside of machine-learning that can produce a solve-swap sequence for any arbitrary permutation graph in polynomial time.[al.b, p. 1] Typically, the placement that the token- swapping problem aims to solve is the identity placement

$$P_I(t_u) = u; u \in G.V.$$

Some approaches may use intermediary placements between an initial placement $G.P$ and the identity placement P_I to find shorter swap-sequences.

3.1.1 Metrics in the Token-Swapping Problem

Before moving on to describing the approaches used in the token-swapping problem, it is important to understand some of the metrics used in the development of the *DelineCrypto* system.

DelineCrypto implements a permutation graph structure through *UTGraph* and uses an additional structure called a *UTGSwapper* that holds a *UTGraph* and determines a solve-swap sequence. The first step taken by the system's *UTGSwapper* is to run a breadth-first search process that determines the shortest pairwise-vertex distances. [al.a] These next definitions are frequently used in explanations related to the *UTGSwapper*.

Definition 3.4 (Subgraph)**.** For a graph G, another graph G_i is a subgraph of G, denoted as $G_i \subseteq G$, if these conditions are true:

- $G_i.V \subseteq G.V$.
- For any edge $e = \{u, v\} \in G.E$ such that $G_i.V \bigcap \{u, v\} = \{u, v\}$, e is also an edge of G_i. Otherwise, $e \notin G_i.E$.

Definition 3.5 (Path)**.** For two non-identical vertices v and u in a graph G, a path between them is a sequence of vertices $S = n_0, n_1, \ldots, n_p$, such that $\{n_0, n_p\} = \{u, v\}$, and all elements of S are unique. The length of the path is p, and $P.E$ and $P.V$ form a subgraph of G. All vertices $v \in P.V$ such that $v \notin \{n_0, n_p\}$ have degrees of two, and n_0 and n_p both have degrees of one.

Remark. A *degree* in the context of graph theory is a measure of a vertex's number of edges. In simple undirected graphs, each pair of unique vertices can only have one edge. A pair of unique vertices that share an edge are *neighbors*.

Definition 3.6 (Shortest pairwise-vertex distance)**.** For two non-identical vertices u and v of G, the length of the shortest path S between u and v, denoted as

$$d(u, v) \to n; n \in \mathbb{N}_{>0}.$$

There may be more than one path between two vertices of a graph. However, unless otherwise stated, the term "distance" when referring to two graph vertices is the length of the shortest path between them.

An important metric used in the token-swapping problem is the token distance. The token distance of a token t_u is computed by a function related to the shortest pairwise-vertex distance.

Definition 3.7 (Token distance)**.** Given a token t_u sitting on a vertex of a permutation graph G, and a token placement P', the distance of t_u is

$$d_t(t_u) \to d(G.P(t_u), P'(t_u)).$$

Definition 3.8 (Cumulative token distance)**.** A measure for a permutation graph G given by the function

$$\Sigma_{v \in G.V} d_t(t_v).$$

Definition 3.6 is important to understanding some of the swap classifications pertinent to the token-swapping problem.

Definition 3.9 (Swap score)**.** For a permutation graph G with a wanted placement P', and an edge $e = \{u, v\}$ used for a swap, the distance delta function for a token sitting on one of the vertices of e is

$$s_f(v) = d(u, P'(t_v)) - d_t(t_v).$$

The output value from s_f is in $\{-1, 0, 1\}$. The swap-score function is then

$$S_f(e) = (s_f(u), s_f(v)).$$

Definition 3.10 (Best swap)**.** A classification of a swap s_e for a permutation graph G and its wanted placement P' that uses the swap-score function S_f. Denote the sum of the two output values of S_f as $T(S_f)$. The edge $e = \{u, v\}$, s_e is the best swap if

$$T(S_f(e)) = \min_{e' \in G.E} T(S_f(e')).$$

Definition 3.11 (Distraction swap)**.** On a permutation graph G with a wanted placement P', a swap s_e is a *distraction swap* if before $s_e(G)$, vertex $v \in e$ is solved, in other words, the condition $P'(t_v) \stackrel{?}{=} G.P(t_v)$ is true, but after $s_e(G)$, $P'(t_v) \neq G.P(t.v)$.

Breadth-first search is the methodology of choice in the system's initial calculations of pairwise vertex paths and distances.[al.a] Algorithms required by the *UTGSwapper* class require knowledge of as many pairwise-vertex paths within the limits designated to the system by the user. *DelineCrypto*'s implementation of breadth-first search,in one of its structures *GInfo*, is extended to accomodate for more than one path (shortest path) between any two connected vertices. *GInfo* uses a *BasicSearchStruct* that is a referential information container for a breadth-first search operation starting from an arbitrary source vertex to all other connected vertices. This structure holds the information

- vertex identifier
- parent identifier
- distance.

GInfo holds a map

$$M : v \to (u \to \texttt{BasicSearchStruct}); u, v \texttt{ are vertices.}$$

The input values of M are the source vertices, and each output value is a map of each vertex to a sequence of their *BasicSearchStruct*'s ranked by the distance variable (with respect to the source vertex that is the input value to M). The map M is designed to hold the minimal amount of path information obtained from breadth-first search while also capable of being used to "backtrace" for pairwise-vertex paths.

In typical breadth-first search operations for a selected source vertex $s \in G.V$ (G the permutation graph), for every level V_l that consists of a set of neighboring vertices travelled from the previous vertex n, the algorithm stores only the shortest distance and parent vertex to the source vertex s. But *GInfo*, given a user-specified positive integer *DEFAULT_MAX_PARENT_SIZE*, records some $r' \leq r$ (shortest distance,parent vertex) pairs for each vertex $v_i \in G.V$. And when *GInfo* has to output the paths for a pair of vertices $u, v \in G.V$, it uses a user-specified variable *DEFAULT_MAX_PATH_SEARCH_SIZE* to restrict the number of paths it outputs. This variable is an important limit, especially in graphs of vertex size greater than 50 (on a typical personal computing system). *DEFAULT_MAX_PATH_SEARCH_SIZE* works by restricting the number of sub-paths in its cache from the source vertex v to all other connected vertices. Each additional subpath or extended subpath is inserted at the location of its rank by distance among the other paths in the cache, and the cache is trimmed to satisfy this user-specified limit. These two variables help to ensure that path-finding does not exceed the time and space resources of the user's computing hardware, because it is believed by many that the problem of finding all paths between two connected vertices in a graph is NP-Hard.[Stab] [Staa]

The limitations of path-finding implementation in *DelineCrypto* increases the probability of some inconsistencies in shortest pairwise-vertex paths. Exemplifications of this deficit include the following:

- Two vertices $u, v \in G.V$, G a graph, may each have ordered sequences of paths between them in *GInfo* that are different.
- For three vertices $u, v, x \in G.V$, such that u and v are neighbors (edge distance of 1), the ordered sequence of paths to x from u and that of x from v may not include each other in their shortest paths. So deriving the distance of x from u using the paths with x as the source may require additional search operations.

Additionally, there is only one breadth-first operation called right before token-swapping operations take place, so operations that may require focusing

only on *subgraphs* (see Definition 3.4) of some permutation graphs might not have access to "perfect information" regarding shortest paths that exclude some set of vertices from travel. The breadth-first search operation by *GInfo* takes about

$$O(|G.V| + C|G.E|).$$

The multiplier is the variable *DEFAULT_MAX_PARENT_SIZE*. But the breadth-first search process merely produces possible parent vertices for a target vertex to the source vertex. To find the set of possible paths using the knowledge of parents, *GInfo* takes a time of

$$O(C^{|G.V|}).$$

This paper now moves on to another important concept, subgraph decomposition, that is used in *DelineCrypto*'s solving of the token- swapping problem.

Definition 3.12 (Subgraph decomposition)**.** A partitioning of a graph G into a set of m subgraphs $S_G = \{G_0, G_1, ..., G_{m-1}\}$ with these properties:

1. For any two subgraphs $G_i, G_j \in S_G$,

$$G_i.V \bigcap G_j.V = \varnothing.$$

2. The union of the subgraph vertices of S_G is equal to the set of vertices $G.V$.

Two concepts, defined in Definition 3.8 and 3.9, relating to graphs are important in calculating subgraph decompositions in the *DelineCrypto* system.

Definition 3.13 (Eccentricity)**.** A measure of a vertex $v \in G$, a simple undirected graph, defined as

$$ecc_G(v) = \max_{u \in G.V} shortest\_distance(u, v).$$

Definition 3.14 (Center)**.** Given a simple undirected graph G, the set of vertexs $C \subseteq G.V$ that satisfies the condition

$$\forall c \in C, ecc_G(c) = \min_{v \in G.V} ecc_G(v).$$

One important observation of the token-swapping problem relates to the bounds of k, the length of the swap-sequence to solve for an arbitrary token placement on a permutation graph.

Lemma 3.1. *The bounds of the shortest solve-swap sequence k for a token placement P' in an unsolved permutation graph G is based on the cumulative token distance c of G with respect to P', and should be expected to fall in the bounds*

$$\lfloor \frac{c}{2} \rfloor \leq k < c + |G.V|.$$

Lemma 3.1 is quite useful because the question of the existence of a solve-swap sequence $\overrightarrow{S}$ for permutation graph G of length $k < \frac{c}{2}$, c the cumulative token distance, can be answered with a definitive "does not exist". However, there is a probabilistic element to this lemma because the upper-bound is not guaranteed to be satisfied for all permutation graph cases.

3.2 Approaches to the Token-Swapping Problem

To remind, the main objective of *DelineCrypto* is to act as a system for encoding characters, such that some encoding schemes can be deemed "cryptographically secure".

The approaches to solving the token-swapping problem are static in nature. If a *UTGSwapper* instance was to be given the same arguments to find a solve-swap sequence for a permutation graph, it would consistently output the same sequence for any attempt. So the methodologies used to enable this component of the system to learn from its mistakes does not exist.

There are three categories of swap methodologies that *UTGSwapper* can take, all of which require an initial breadth-first or depth-first search to obtain pairwise path and distance information. These categories are named *RSwap* (route swap),*ObjSwap* (objective swap), and *TNSwap* (target node swap). The first category is the most computationally expensive.

3.2.1 The token-swap methodology RSwap

RSwap, shorthand for the route-swap methodology to solve the token-swapping problem, relies on the information obtained in the initial breadth-first search. There are two steps to this methodology.

1. Subgraph decomposition of permutation graph G, and re-arrange the sequence of subgraphs into the "best" order that they are to be solved.

2. Solving each subgraph $S_G \in G$ by first ordering the vertices $S_G.V$ into V_{S_G}, and then solving each vertex $n \in V_{S_G}$ by swapping its solve-token along a path discovered by breadth-first search.

Subgraph decomposition goes by a particular sequence of instructions. These are the broad steps that *DelineCrypto* takes to produce a subgraph decomposition of target size d, a user-specified positive integer, on an arbitrary graph. The algorithm next described produces a decomposition of size roughly equal to d',

$$\lceil d \times 0.5 \rceil \leq d' \leq \lceil d \times 1.5 \rceil.$$

This is an important formula used in decomposing each component. It calculates the mean distance between a vertex v and a non-empty set of vertices V' in a connected graph G.

$$D_m(v, V') = \Sigma_{u \in V'} d(u, v)/|V'|.$$

Subgraph decomposition

1. Given a connected graph G, set the variable $S_D = \{G.V\}$ as the running solution.
2. If $|S_D| \geq d$, algorithm terminates.
3. Choose the element $V_S \in S_D$ with the largest number of vertices, and remove V_S from S_D.
4. Assign S_G as the subgraph of G corresponding to V_S. Calculate the center C_{S_G} of $S_G.V$.
5. Assign

   $S_D^{(2)} = \{V' | \texttt{V' a set of vertices to a connected component of the subgraph } S_G.V - C_{S_G}\}$.

   $S_D^{(2)}$ is the set of all components of S_G excluding its center.
6. Terminate the algorithm if the below condition is true:

$$|(|S_D| + |S_D^{(2)}| - d)| \geq |(|S_D| + d)|.$$

7. Add each vertex $v \in C_{S_G}$ to one set of vertices in $S^{(2)}$ by this decision function:

$$D_{S_G}(v) = \min_{V' \in S_D^{(2)}} D_m(v, V').$$

8. Add $S_D^{(2)}$ to the running solution:

$$S_D \leftarrow S_D \bigcup S_D^{(2)}.$$

   .
9. Loop back to steps 2-8.

The user can specify any positive integer d, the number of wanted subgraphs. However, one of these values is advised for an adequate solution on a graph G if the optimal value of d is not known.

- $\max_{v \in G.V} deg(v)$.
- $|G.V| / \max_{v \in G.V} deg(v)$.

The subgraphs S_D are rearranged in descending order by a mean center distance measure based on the previously described function D_m.

$$SORT(S_D) = \texttt{argsort}_{S_D^{(2)} \in S_D}(\Sigma_{c \in C(G)} D_m(c, S_D^{(2)}) / |S_D^{(2)}.V|).$$

RSwap, after ascertaining a sorted subgraph decomposition S_D of permutation graph G, proceeds to iterate through each of the subgraphs and solve the wanted placements of their vertices.

The crucial problem for solving each subgraph is to find a sequence of paths that "routes" all of the subgraph's tokens to their wanted placement, in effect solving the subgraph.

Definition 3.15 (Token route)**.** For a wanted placement P', an *uninterrupted* sequence of swaps along a path $P \subseteq G$, a permutation graph, that transfer token t_u to the vertex that is the wanted placement $P'(t_u)$.

It is important to define "token route" by Definition 3.14 to specify that it is an uninterrupted sequence of swaps using path P, instead of intermittently swapping the i'th edge of P, conducting a swap not of P, and then continuing on to swapping the $(i+1)$'th edge.

The best solution consists of the following elements:

- The best order of the vertices in the subgraph.
- The best route for each of the vertices in that order.

For a subgraph S_G of vertex size n, searching for this best solution entails considering all $O(n! \times n^2)$ possibilities, if no other information is used in the process of elimination. The value $n!$ is the possible orderings of the vertices, and for each ordering, there is an upper bound of n^2 paths (consider $n-1$ routes for each vertex). With this high runtime in mind, *another solution is considered that may not produce the best solution*, but one that performs "relatively well". The algorithm "subgraph route-search" uses a cache C throughout the entire *RSwap* process that stores all vertices that have been "solved" for a wanted placement. It also uses a "path intersection" function of the form

$$I_P(P, C) = |P \bigcap C|.$$

Subgraph route-search algorithm

- For a subgraph S_G corresponding to a subset $V_S \subseteq G.V$, order the vertices $v \in V_S$ by descending order using function $D_m(v, C_{S_G})$, where C_{S_G} is the set of vertices comprising the center of S_G.
- For each $v \in V_s$:

  [-] Get the possible routes $\overrightarrow{R}$ for token sitting on $G.P(t_v)$ to v. Determine the best path (route) by

  $$P_{best}(R, C) = argmin_{r \in R} I_P(r, C).$$

  [-] Conduct the swap sequence $P_{best}(R, C)$.

  [-] Add v to cache C.

The above subgraph route-search algorithm assumes the "best" ordering of vertices is based on the vertex distance to the center. For each vertex v in this order, it determines which path in the set of known paths from v to $G.P(t_v)$ produces the minimal distractions. The *subgraph route solution* can be explained with the aid of this algorithm.

Subgraph route solution

- For a subgraph S_G from $V_S \in G.V$, first run the *subgraph route-search* algorithm on S_G.
- Using the order of V_S, loop this next step until solved.

  [-] Iterate through V_S (left to right) and find an unsolved vertex $v \in V_S$. If v does not exist, then terminate. Otherwise, continually conduct *best swaps* on token t_v until v is solved.

*One important concept that **RSwap** uses is "farthest from the center, nearest from a center".* This means that given a subgraph decomposition for a permutation graph G, first choose the farthest unsolved subgraph from the center of G to be solved, then solve those vertices of the subgraph in the order of their least to greatest distance from the center. Next is a proof that demonstrates this concept is critical to finding a swap sequence of minimal length for a subset of the token-swapping problem, defined in Definition 3.10, although the exact procedure by *RSwap* seldom finds swap sequences of minimal length in graphs that are not orthodox (paths, regular graphs, complete graphs, etc.).

Definition 3.16 (Token-swapping by routing.)**.** A subset of the token-swapping problem (Definition 3.3) in which a swap sequence $\overrightarrow{S}$ must swap using as many pairwise-vertex paths as possible.

Proof. Assume the permutation graph G is a connected graph, and subgraph decomposition produces S_D. Then for each of the subgraphs S_G of $S_D^{(2)} \in S_D$, solving S_G by *subgraph route solution* excludes the vertices $S_G.V$ from being part of a swap operation for the subgraphs after to be solved, effectively eliminating it from the possibility of being unsolved in future swaps. This fact is due to a few reasons:

- If S_G is at the same distance from the center of G as another subgraph $S_G^{(2)}$, then there must exist a set of paths from those solve tokens of $S_G.V$ to the vertices $S_G.V$ that do not require involving the vertices of $S_G^{(2)}$ in a swap. A parallel argument can be made for $S_G^{(2)}$.
- Solving each subgraph S_G swaps out the tokens that solve the vertices $G.V - S_G.V$. The new locations of these tokens sits on vertices that have not been attempted to be solved. And the distances of these tokens to their wanted placement is of lesser distance than before solving for S_G.

Another important point is the ordering of the vertices of S_G to be solved by increasing distance to the center minimizes the probability that the "best" path for a vertex in that order will unsolve a previous vertex by colliding with it. ■

The explanation to the procedure of *RSwap* lends way to proving Lemma 3.1.

Proof. Observe that the minimal swap-length bounds for an arbitrary permutation graph G with the cumulative token distance c is$\lfloor \frac{c}{2} \rfloor$. The line graph, otherwise known as a path, which consists of a set of vertices V of size greater than 1 such that two vertices have degree 1 and the rest have degree 2, satisfies this value by *RSwap*. Line graphs are thus the most basic and easiest permutation graphs to be solved.For any permutation graph G, there are two important aspects to consider.

- The total number of distraction swaps in solving each subgraph.
- The distances of the tokens sitting on a subgraph S_G before and after solving S_G.

If there are no distraction swaps, then the length l of the solve-swap sequence for G satisfies the condition

$$l \leq c.$$

But suppose distraction swaps occur running the *subgraph-route search algorithm* on an arbitrary subgraph $S_G \in G$. *Then there are a maximum of* $|S_G.V|$ *extra swaps by the "best" decisions.* By this claim, summing up these extra swaps of every subgraph results in a maximum of $|G.V|$ extra swaps to fix those decision swaps. However, the lack of guarantee in the upper bound posed by Lemma 3.1 comes down to the sub-optimal swaps used in routing each token. The term "sub-optimal swap" is used to denote a swap with score $(1, -1)$ or $(-1, 1)$. A sub-optimal swap increases the cumulative token distance of the permutation graph. The assumption of Lemma 3.1 is that the ratio of sub-optimal swaps over swaps with scores of $(-1, -1)$ is approximately 1, thus negating the penalizing effect on the cumulative token distance. ■

This section on *RSwap* concludes by describing some classifications of path pairs based on collision.

Definition 3.17 (Opposing paths)**.** Given two paths $P_0, P_1 \in G$, a graph, that are generically represented as

$$P_0 =< v_1^{(0)}, v_2^{(0)}, \ldots, v_{p_0}^{(0)} >,$$

$$P_1 =< v_1^{(1)}, \ldots, v_{p_1}^{(1)} >; p_0, p_1 \geq 1,$$

P_0 and P_1 oppose each other if there exists a subpath

$$< v_{x_1}^{(i)}, \ldots, v_{p_i}^{(i)} >=< v_{p_j}^{(j)}, \ldots, v_{p_j-(p_i-x_1)}^{(j)} >; \{i, j\} = \{0, 1\}, 1 \leq x_1 \leq p_i.$$

Definition 3.18 (Uni-directional paths)**.** Using the path notation from Definition 3.14, two paths P_0 and P_1 are uni-directional paths if for one of the paths P_i, there is a subgraph

$$P_i^{(1)} =< v_{x_1}^{(i)}, \ldots, v_{p_i}^{(i)} > \subseteq P_i$$

such that $P_i^{(1)} \subseteq P_j$,$\{i, j\} = \{0, 1\}$.

Definition 3.19 (Colliding paths)**.** A subclass of uni-directional paths (see Definition 3.15) such that the shared subgraph is a single vertex, an endpoint of one of the paths.

Definitions 3.16-18 provide a good classification scheme for algorithmic improvements in vertex-order and pairwise-vertex path selection in *RSwap*. Algorithmic improvements in *RSwap*. An important observation is that the problem of *token-swapping by routing* (Definition 3.14) restricts itself to primarily swapping each token along their **entire selected path** for the token's wanted placement. This restriction means that sub-optimal swaps are inevitable in some graph structures.

Definition 3.20 (Sub-optimal token-routing situation)**.** In the token-swapping problem, a *sub-optimal* swap is unavoidable if for two vertices u, v, there does not exist paths P_u, P_v (respectively for vertices u and v) that do not result in a distraction swap using either order of P_u and P_v.

One suggested improvement for paths of classification defined in Definitions 3.15-17 is path-splitting. A path P_0 for a token is split into two subgraphs $P_0^{(0)}$ and $P_0^{(1)}$. For another path P_1 that would produce the sub-optimal token- routing situation, by path-splitting, the route sequence then becomes $P_0^{(0)}, P_1, P_0^{(1)}$.

3.2.2 The token-swap command ObjSwap

ObjSwap stands for objective swap, based on the swap-score function (see Definition 3.8). The procedure for this command, similar to *RSwap*, requires information from a breadth-first search operation by *GInfo*.

When *ObjSwap* is called, it is given a number of arguments, one of which is pertinent to providing it with a minimal description in this section. That argument is named "best" (a boolean).

For a permutation graph G, the set of edges $E' \subseteq G.E$ that are the best swaps (see Definition 3.9) are

$$S_b(G) = \{e | e \in G.E \texttt{ and } T(S_f(e)) = min_{e' \in G.E} T(S_f(e'))\}.$$

Similarly, if "best" is set to "False", then the swaps to consider are

$$S_w(G) = \{e | e \in G.E \texttt{ and } T(S_f(e)) = max_{e' \in G.E} T(S_f(e'))\}.$$

Using a selector $rand_{sel}$ that selects an element in a set by pseudo-random mechanisms, the $OBJSWAP$ can be described as such:

$$OBJSWAP(G) = \begin{cases} rand_{sel}(S_b(G)) \texttt{ if "best" = "True",} \\ rand_{sel}(S_w(G)) \texttt{ otherwise.} \end{cases}$$

This selector is implemented by an *AbstractPRNG* instance in *DelineCrypto*. There is one important observation to make about this decision scheme for solving the token-swapping problem. Denote the set of all solve-swap sequences that set *ObjSwap*'s argument "best" to "True" as $GREEDY\_DECISIONS$. Then there is a possibility that the swap-sequence $\overrightarrow{S} \notin GREEDY\_DECISIONS$ is the shortest swap sequence.

Proof. The proof proceeds by generalization of situations that result in *distraction swaps* and other low-scoring swaps. Suppose that a swap sequence $\overrightarrow{S} \in GREEDY\_DECISIONS$ is selected, with size $r = |\overrightarrow{S}|$. At the i'th index, for $0 \leq i < r - 1$, the edge $e = \{u, v\}$ is chosen with one of the following scores: $(-1, 1)$, $(1, -1)$, or $(-1, -1)$. A swap s_e is conducted on the chosen edge e because it is one of the *best swaps*. But in doing so, the only available *token routes* for one of the vertices of e, u or v, that do not use the swap s_e anymore (otherwise, the swap s_e turns into an extra swap), result in these changes in cumulative token distance:

- k_1 distraction swaps on the path P_1, so there needs to be around k_1 swaps to solve those distraction swaps,
- for the remaining set of tokens T' that did not become "distracted" but also displaced from their original vertex locations, there are $T'_0 \subseteq T'$ that have to swap on "best" swaps of score $(-1, 1)$, in effect meaning that each of the other tokens during those swaps require one additional swap to be solved. In this situation, there are $O(|T'_0|)$ additional swaps that need to be made. These tokens T'_0 could have used swaps that result in less cumulative token distance if the path P_1 was not selected.

Conversely, if a lower-scoring swap e' was selected, such that $e' \bigcap e = \{u\}$, then for any available path for swapping vertex u to its wanted placement requires at least one additional swap than if e was selected and the swaps on path P_1 were taken. But there are k_1 less distraction swaps and $O(|T'_0|)$ swaps that did not need to be taken. This means that there exists a solve-swap sequence S', $S' \neq S$, such that

$$|S'| = r + 1 - k_1 - O(|T'_0|) + c < |S|; c \in \mathbb{N}_{>0}.$$

The non-negative integer c represents the size of the set of swaps that are taken after $s_{e'}$ that would not have been if s_e was instead taken. ■

One important observation about *ObjSwap* is that it can result in an infinite swap sequence, depending the specific $rand_{sel}$ used, for example, continually choosing the edge e that alternately scores $(1, -1)$ and $(-1, 1)$.

3.2.3 The token-swap command TNSwap

The "target node" swap (*TNSwap*) command takes as input a token t_v of a permutation graph, and chooses a path P in *GInfo* that would place t_v on vertex v if P is used as a route. It then swaps t_v on P.

4 Commands and Expression in *DelineCrypto*

Before discussing the two file classes of central importance, *RInst* and *Dinst*, there is one important structure in the *DelineCrypto* system used, and that is the *DMDTraveller*, that operates on a multi-dimensional delineation constructed by a *DelineMD* instance. The autonomous activity of the *DMDTraveller* is dictated by the coupled instructions found in its corresponding *DInst* and *RInst* files.

It is important to note that many of the specifications in this section are arbitrarily designed, so there is room for modification (deletion, extension, addition) that would not fundamentally alter the autonomous activity of the *DelineCrypto* bot.

4.1 The Operative *DMDTraveller*

DMDTraveller can be thought of in the two main aspects that it acts, as a pointer on a *DelineMD* instance and as a dataset modifier. Recall from the earlier section on delineation, that every *Deline22* for a specific *Deline22MC* holds a different subdata to its initial three-dimensional dataset. All *Deline22* instances of the D_{md} will have their index pointers (to their datasets) at 0.*DMDTraveller* uses an abstraction of a pseudo-random number generator called *AbstractPRNG* to aid it in selecting choices at some decision junctions.

4.1.1 The *AbstractPRNG* structure

The *AbstractPRNG* structure stands for "abstract pseudo-random number generator". It is so named not because any arguments that it is given enables it to output a pseudo-random numerical sequence but because the intended use by *DelineCrypto* in some of its processes such as *DMDTraveller* require pseudo-random number generators that allow it to make autonomous decisions that cannot be so easily predictable by a user and/or the user's machine.

In fact, in many cases, a *AbstractPRNG* instance outputs a relatively "predictable" sequence of numbers that fail randomness tests. So a user that instantiates a *AbstractPRNG* must ensure for their own secrecy that that the generator's output is not so easily predicted by an observer with some arbitrary collection of background knowledge on the generator. The generator simply outputs some arbitrarily- sized (usually infinite) sequence of numbers in the spaces $\mathbb{N}$ or $\mathbb{R}$.

There are a wide variety of randomness tests that have been designed by researchers for computational use. For example, two effective tests are the gcd test and the Gorilla test.[MT02] There are few advanced procedures implemented in

the *DelineCrypto* system that use techniques as defined or rigorous as those two tests mentioned. There is one structure called the *APRNGGauge* that can serve as the unit towards building more advanced randomness tests. *APRNGGauge* observes two aspects of a sequence of numbers from a *AbstractPRNG*.

- coverage of the sequence over a finite range I.

- pairwise-point distance of the sequence over I.

For measuring coverage of a numerical sequence $\overrightarrow{S}$, a numerical two-tuple I, representing a range, such that $I[0] \leq I[1]$ and a decimal point radius r is used. See the $COVERAGE$ function that essentially calculates the unique area that the points on $\overrightarrow{S}$ span on I.

Remark. The value of r is an important determinant in the measure of coverage, and is usually calculated in proportion to the range I.

Remark. The range I should contain all elements of $\overrightarrow{S}$.

---

function $AdjustRange(rx,I)$
  if $rx[0] < I[0]$ **then**
    $rx[0] \leftarrow I[0]$
  end if
  if $rx[1] > I[1]$ **then**
    $rx[1] \leftarrow I[1]$
  end if
end function
return rx

```
function NonIntersectingActivationRanges(S⃗, I, r)
                                   ▷ sort unique elements in ascending order
    S⃗_0 ← S⃗.unique().sort()
    R⃗ ←< (S⃗_0[0] − r, S⃗_0[0] + r) >
    R⃗[0] ← AdjustRange(R[0], I)
    for i = 1; i < |S⃗_0|; i++ do
        rx ← (S⃗_0[1] − r, S⃗_0[1] + r)
        rx ← AdjustRange(rx, I)
        if S⃗_0[−1][0] ≤ rx[0] ≤ S⃗_0[−1][1] then
            rx[0] ← S_0[−1][1]
        end if
        if S⃗_0[−1][0] ≤ rx[1] ≤ S⃗_0[−1][1] then
            rx[1] ← S_0[−1][1]
        end if
        R⃗.append(rx)
    end for
    return R⃗
end function
```

```
function COVERAGE(S⃗, I, f)
    R⃗ ← NonIntersectingActivationRanges(S⃗, I, f)
    s ← Σ_{r∈R⃗}(r[1] − r[0])
    return s
end function
```

As for the pairwise-point distance measure, it is called the "unidirectional weighted point distance" measure in *DelineCrypto* because it is not the absolute distance between points but a ratio of the weighted distances between points. The distances are weighted by the index difference between the real numbers in the numerical sequence of interest. The algorithm is the function $UWPD$.

```
function PointDistanceMeasure(S⃗, i)
    d ← 0
    for j = i + 1; j < |S⃗|; j++ do
        d ← d + ‖S[i] − S[j]‖ /(j − i)
    end for
    return d
end function
```

```
function CumulativePointDistanceMeasure(S⃗)
    d ← Σ_{i∈[0,|S|−1]} PointDistanceMeasure(S⃗, i)
    return d
end function
```

```
function MaxPointDistanceMeasure(I, l)
    S⃗ ←<>
    i ← 0
    for j = 0; j < l; j++ do
        S⃗.append(l[i])
        i ← (i + 1) mod 2
    end for
    return CumulativePointDistanceMeasure(S⃗)
end function
```

```
function UWPD(S⃗, I)
    d_0 ← CumulativePointDistanceMeasure(S⃗)
    d_1 ← MaxPointDistanceMeasure(I, |S|)
    return d_0/d_1
end function
```

Remark. Note that $UWPD$ will always produce a real number in the range $[0, 1]$.

The two measures of *AbstractPRNG* exemplified by the previous pseudocode allows a numerical sequence to be easily understood in terms of its variance over a possible range of values (coverage) and the relative differences between any two values of the sequence with respect to position (uni- directional weighted point distance). For example, a sample sequence $\overrightarrow{S}' = \overrightarrow{1} \times c$, $c \in \mathbb{R}$, has a $UWPD$ score of 0 and a coverage on the scale of some negligible fraction $\frac{1}{x}$, $x \in \mathbb{R}_{>0}$ (depending on the point radius r).

DelineCrypto comes equipped with a few numerical generators. For purposes of further development, **every generator of the class *AbstractPRNG* should have the capability to be instantiated by a stringized representation of its arguments.** This paper does not provide algorithmic descriptions of these generators, but here is a list of the ones readily available (as of August 2023).

- standard random generator (StdRandGenerator)
- linear congruential generator (LCG)

- two-headed linear congruential generator (LCG2H)
- permutative linear congruential generator (PermLCG)
- value-index stretch generator (ValueIndexStretchGenerator).

Remark. For more information on the specific templates to represent each of these generators as a string, refer to the actual computer code.

Every type of *AbstractPRNG* uses two main functions to output values. One function outputs values without any bounds specified, and the other does so in a specified range. **All types of *AbstractPRNG* should be designed to be deterministic.**

Because *AbstractPRNG* is merely a templated structure, extending this structure to provide the user with more obscure and unorthodox numerical generators may prove beneficial. A personal observation on the generators in the above list is that linear congruential generators are "predictable" over some relatively small and finite number of cycles (a cycle starts and ends when the output is greater than the modulo). Permutative linear congruential generators are used in the particular decision problem of arbitrarily re-ordering a sequence of elements.

Remark. In Section 5, the reader can gather how "strong" pseudo-random number generators make great use in automating instructions and other tasks involved in instantiating *DelineCrypto*'s encryption process.

The numerical values of an *AbstractPRNG* instance are used by *DelineCrypto* as either literal values (directly as they are) or as element selectors in a sequence (with the use of the **InRange* functions). The bijectivity of the mapping between the space of possible *AbstractPRNG*s and the space of numerical sequences is difficult to quantify. For unbounded output values, assume that a typical generator of arbitrary type A belonging to the *AbstractPRNG* class produces a non-identical sequence of numerical values than any other generator of A. But two generators T_1 and T_2 of types A and B have a non-zero likelihood of producing identical numerical sequences. So the map

$$GENSPACE : SPACE(AbstractPRNG) \to SPACE(\texttt{numerical sequences})$$

is not bijective, but the number of unique *AbstractPRNG* instances that produce a particular numerical sequence is arbitrary (no quantification).

4.1.2 The command "Move One"

The "Move One" command is a templated comma-separated string that tells a *DMDTraveller* how to move its pointer, of this form

(0) MO

(1) "number of iterations"

(2) "*Deline22MC* index"

(3) "*Deline22* index"

(4) "direction sequence"

(5) "ratio sequence"

(6) "boolean on assignment to closest point".

Elements 1-3 are integers, and additionally, element 2 is adjusted via a modulo operation and possible +1 shifts for a valid index (input) in the map $D_{md}.H$. And element 3 is modulo the number of *Deline22* instances for that specific *Deline22MC*. The forms for elements 4 and 5 respectively are

$$(l|r|t|b); \texttt{separated by "."}$$

and

$$float; \texttt{separated by "\_"}.$$

Elements 4 and 5 are of the same length, and each i'th value of element 5 corresponds to the ratio coupled to the i'th element of element 4.

Index 6 determines whether the acquired point from the MO iteration should be rounded to the closest point on the dataset of the *Deline22*.

An example "Move One" command is

`MO,3,2,3,l.r.t.b.b.r,0.5_1_0.75_0.5_1_0.2,1.`

"Move One" produces a sequence of points that the *DMDTraveller* has travelled on by its instruction. The command is essentially a cumulative sum of a matrix m with the number of rows equal to the length of element 4 to the data point at the index of a *Deline22*.

This next function calculates a two-dimensional point on a line represented by two-dimensional points p_0 and p_1 by an axial ratio value q, typically in the range $[0, 1]$ but certainly any number $q \in \mathbb{R}$ would do.

Remark. The reason why the term "axial ratio" is used instead of "ratio" is because the value q is a fractional value on only one of a line's axes, the x-axis if the line is not vertical, instead of the line's Euclidean distance.

---

function $P_{ar}(p_0,p_1,q)$
  $l \leftarrow line(p_0, p_1)$
  if $l.m = NAN$ **then**
    return $(p_0[0], p_0[1] + q \times (p_1[1] - p_0[1]))$
  else
    $x \leftarrow p_0[0] + q \times (p_1[0] - p_0[0])$
    return $(x, x \times l.m + l.b)$
  end if
end function

Another important function is "nearest edge point", which calculates the closest point to some arbitrary point p on a *Deline22*'s edge.

function $TO\_SEQ(L_F, d)$
  $\overrightarrow{F} \leftarrow L_F[d]$
  ▷ convert the sequence of fits to a sequence of points
  $\overrightarrow{P} \leftarrow < \overrightarrow{F}[0][0], \overrightarrow{F}[0][1] >$
  for $(i = 1; i < |\overrightarrow{F}|; i++)$ **do**
    $\overrightarrow{P}.append(\overrightarrow{F}[i][1])$
  end for
  return $\overrightarrow{P}$
end function

function $P_{ep}(p, L_F, d)$
  $\overrightarrow{P} \leftarrow TO\_SEQ(L_F, d)$
  return $argmin_{i \in [0, |\overrightarrow{P}|-1]} \left\| p - \overrightarrow{P}[i] \right\|$
end function

function $MOVE(D_{md}, x_2, x_3, x_4, x_5, x_6)$
  $L_F \leftarrow D_{md}[x_2, x_3]$
  ▷ get closest index for each edge
  $DIR \leftarrow \{l, r, t, b\}$
  $I_m \leftarrow []$
  ▷ get the two-dim. data point in *Deline22*
  $p \leftarrow L_F.D.row(L_F.i)$
  for $dx \in DIR$ **do**
    $I_m[dx] \leftarrow P_{ep}(p, L_F, dx)$
  end for
  ▷ iterate through the directions
  $M \leftarrow <>$
  for $i = 0; i < |x_4|; i++$ **do**
    $\overrightarrow{P} \leftarrow TO\_SEQ(L_F, x_4[i])$
    $p_1 \leftarrow P_{ar}(p, \overrightarrow{P}[I_m[x_4[i]]], x_5[i]$
    $I_m[x_4[i]] \leftarrow (I_m[x_4[i]] + 1) \mod |\overrightarrow{P}.rows|$
    $M.append(p_1)$
  end for
  ▷ add cumulative sum of M to p
  for $i = 0; i < |M.rows|; i++$ **do**
    $p \leftarrow p + M.row(i)$
  end for
  $L_F.i \leftarrow argmin_{i \in [0, |L_F.D.rows|-1]} \left\| p - L_F.D.row(i) \right\|$
  if x_6 **then**
    $p \leftarrow L_F.D.row(L_F.i)$
  end if
  return p
end function

Remark. MOVE outputs a two-dimensional point, and the function can be understood as a delta function given a point argument p. The delta uses the "attractive" pull/push of edges by direction, initially closest to p and then incremented for each additional call to that direction.

The *MOVE* function above is run by the "Move One" command for the number of iterations specified by the command's element 1. A *DMDTraveller* instance stores every two-dimensional matrix M into its memory, what can be understood as the "contextual point information" for each point output from *MOVE*. **This contextual point information also serves as the *private cipher* used by *DelineCrypto* in its security protocol.**

4.1.3 Background commands

There is a category of commands with the start element "BG" that stands for "background command", instructions that modify the information that a

specific *Deline22* holds. Two dataset modification schemes were carefully selected for "BG": "ALTER" and "JITTER". The "ALTER" scheme changes the points of the edges of a *Deline22*, effectively altering its specific geometric space of classification. The likelihood that a point has a different classification after an "ALTER" command is

$$Pr[CLASSIFY\_POINT_t(L_F, p) \neq CLASSIFY\_POINT_{t+1}(L_F, p)] \propto$$

$$|AREA(L_F^{(t)}) - AREA(L_F^{(t+1)})| \ \& \ \frac{1}{\left\|CENTER(L_F) - p\right\|};$$

L_F `is the` *`Deline22`* `instance,`

`and the variables t and t + 1 denote`

`the times before and after ALTER.`

And the "JITTER" scheme applies delta values to a portion of the points that the *Deline22* holds. The format of the string commands "ALTER" and "JITTER" is

(1) BG

(2) ALTER ?? JITTER

(3) "number of iterations"

(4) "*Deline22MC* index"

(5) "*Deline22* index"

(6) "minimum param_maximum param"

The *DMDTraveller*'s *AbstractPRNG* attribute aids the algorithms for these background commands to modify the wanted target (dataset or delineation). For more information on the specific programming involved in the "ALTER" and "JITTER" commands, refer to the file "dinst.cpp" of *DelineCrypto*.

4.2 The *DInst* file

The *DInst* file contains a sequence of structures named *DInstSeq*. And *DInstSeq* are, in turn, sequences of *DInst* instances. *DInst* stands for "delineation/data instruction". A *DInst* instance represents a command of one of the following categories: "MO", "BG", "HOP", and "MSK".

The first two categories "MO" (short for "Move One") and "BG" (background command) have been discussed. To summarize on the "MO" category, a *DInstSeq* instance tells a *DMDTraveller* to output a sequence of points. Every call to the *MOVE* function outputs two real numbers. These two-tuples are concatenated to form a two-dimensional point sequence $\overrightarrow{P}$ which then is operated on by the mask commands. After applying the mask commands of a

DInstSeq, the output is a modified version of $\overrightarrow{P}$. **For the next subsection on the *RInst* file, for a *DInstSeq* d_{seq}, this output, the modified $\overrightarrow{P}$, is referred to as its point sequence.**

The last two, "HOP" and "MSK", will not thoroughly explained in this paper, only briefly touched on.

The "HOP" command tells the *DMDTraveller* to alter the index of a specified *Deline22* instance. The template for this string command is this.

- HOP
- "index of *Deline22MC*"
- "index of *Deline22*"
- "string representation of a *AbstractPRNG*"

The "MSK" category stands for "mask functions", and there are three masks available: FOX,SNAKE, and HAWK. The intention behind these masks is to provide a layer of obfuscation over a two- dimensional matrix $\overrightarrow{P}$ output from "MO" commands to *DMDTraveller*. **In order for the user to determine the specific complexity of a "MSK" command, the user has to conduct their own independent analysis of the effects of that command on the output of numerical values.**

The FOX mask, unlike the other two masks, uses the "contextual point information" stored by the *DMDTraveller* in the output of its "Move One" command/s to produce deltas on the original point sequence by linear combinative means. It is named as FOX because of the impressive capabilities of foxes, in their natural habitats, to be swift and specialized in directional changes when they hunt or are hunted. Hence, the FOX mask obfuscates an original point sequence it operates on with the intention of "throwing" an attacker off a logical chain of reasoning about the command behind the point sequence of interest.

The SNAKE mask is so named because it obfuscates a point sequence by applying repetitive (concatenating duplication), permutative, and palindromic operations on the point sequence. The resulting output from a SNAKE mask resembles a snake's trail.

And the HAWK mask relies on arithmetic operators to modify a point sequence. The difference of a point sequence after it goes through a HAWK mask is likened to the rapid change in direction, altitude, and velocity that hawks take in their pursuit of prey.

All of the *DInst* categories have been mentioned. A *DInstSeq* essentially uses a vector of *DInst* instances to output a sequence of points.In the "DInst" file, every contiguous sequence of non-empty strings, each representing a *DInst* instance, is used to instantiate a *DInstSeq*. Every *DInstSeq* in the file is separated by an empty string.

The "MO", "BG", and "HOP" commands are one-apply commands, executed in the order that they were presented in the file. The "MSK" commands, however, are all-apply commands, meaning that they are executed on the point

sequence of the *DInstSeq* after a *DMDTraveller* has run the other categories of commands. If there is more than one "MSK" command, then those commands are run in the order they were presented.

Remark. A *DInstSeq*,in the context of encoding, corresponds to a character in the message alphabet.

4.3 The *RInst* file

The *RInst* file is a supplementary command file to a *DInst* file. The file contains string representations of *RInst* structures. *RInst* structures contain conditional commands, formatted in the style of computer programming languages and produces changes onto a *DInstSeq* instance if its condition is satisfied. *RInst* stands for "reactive instruction" because of the conditions required for it to produce changes on a *DInstSeq*.

Remark. For a typical *RInst* file, for every *DInstSeq*, there is at least one *RInst* instruction that the system considers to possibly "react".

The concept of a reaction as it pertains to *DelineCrypto* is explained more clearly in this definition.

Definition 4.1 (Reaction (in *DelineCrypto*))**.** A sequence of changes that occur to one or both of the permutation graph and *DInstSeq* instances. The instructions of a *RInst* instance are used to produce these changes if the **activation condition** of the *RInst* instance is satisfied.

There are two categories of *RInst* structures, "FOR" and "ANYTIME". The first category is considered only on certain *DInstSeq* instances, while the second category is considered for every *DInstSeq* instance. The big difference between the "FOR" and "ANYTIME" *RInst* commands lies in their first three lines.

These are the templates for the first three lines of "ANYTIME" and "FOR", respectively.

```
ANYTIME
COND "conditiones" W/
"a1,a2,...,aN::(conditional args)"

FOR "character"
IF COND "conditiones" W/
"a1,a2,...,aN::(conditional args)"
```

Remark. The strings in quotes are variables.

In the context of character encoding, "ANYTIME" commands are called for every character, and "FOR" commands are called for the character specified by "character" (in the first line of its template).

The variable following *COND* in the templates above is one of six conditions available in *DelineCrypto*. A user could certainly add more conditions to the program if they wanted to. These six conditions and their conditional arguments (the third lines of the templates) are listed below.

- *IN_BOUNDS_OF_DELINEATION*

  [-] conditional arguments: $m_0\_m_1$; $m_0 \leq m_1$ and $m_0, m_1 \in [0, 1]$.

  [-] example: 0.75_0.9.

  [-] Determines if the ratio of points in the previous point sequence falls within the bounds of the conditional arguments.

- *ARITHMETICSEQ_IN_MOD_RANGE*

  [-] conditional arguments: $\{+|-|/|*\}^n$, mod , range of mod ; $n \in \mathbb{N}_{>0}$

  [-] example: $+--/*+$, 4.3, $[0.9, 2.0]$

  [-] Given a two-dimensional matrix $\overrightarrow{P}$, perform the sequence of arithmetic operators, in that declared order and modulate if needed, on $\overrightarrow{P}.col(0)$ to get some $f \in \mathbb{R}$. Then apply the mod on f and determine if that value falls in the *mod range*.

- *POINTSEQ_INTERSECTS_W_PREVIOUS_SPAN*

  [-] conditional arguments: $m_0\_m_1$; $m_0 \leq m_1$ and $m_0, m_1 \in [0, 1]$.

  [-] example: 0.51_0.79.

  [-] For a *DInstSeq* $d_{seq}^{(1)}$ that has directed a *DMDTraveller* T_d, and the previous *DInstSeq* $d_{seq}^{(0)}$ that has directed T_d, calculate the span of the point sequence $\overrightarrow{P}_0$ of $d_{seq}^{(0)}$ as

  $$< [\overrightarrow{P}_0.col(0).min\,\overrightarrow{P}_0.col(0).max][\overrightarrow{P}_0.col(1).min\,\overrightarrow{P}_0.col(1).max] >,$$

  and determine the ratio of points in $\overrightarrow{P}_1$ of $d_{seq}^{(1)}$ that fall within the span of $d_{seq}^{(0)}$.
  If $d_{seq}^{(1)}$ is the first *DInstSeq* to direct T_d, output "True" by default.

- *POINTSEQ_MULTICLASS_MEASURE*

  [-] conditional arguments: $b\_m_0\_m_1$;$b \in \{0, 1\}$,$m_0 \leq m_1$ and $m_0, m_1 \in [0, 1]$.

  [-] example: 1_0.2_0.3453

  [-] The conditional argument b specifies if the classification type (by the *Deline22MC* instance of a *DMDTraveller* T_d) is hierarchical (1) or child-most (0). Note that the child-most classification has already been described in Section 2. Hierarchical classification is a tree-like ranking of all of the delineations that the point belongs to, such that the child- most delineations are the leaves and the parent-most delineations are the root. Setting b to "hierarchical classification" increases the likelihood that an arbitrary point is multi-classified.

This condition calculates the ratio f of data points, that a *Deline22MC* has delineated on, that have more than one classification and determines if $f \in [m_0, m_1]$.

- *SEQUENTIAL_EUCLIDEAN_DISTANCE_MEASURE*

  [-] conditional arguments: $m_0\_m_1$; $m_0 \in [0,1]$ and $m_1 \in \mathbb{R}$.

  [-] example: 0.33_0.67

  [-] The sequential Euclidean distance of a point sequence $\overrightarrow{P}$ is its cumulative and contiguous pairwise Euclidean point distance.

  The variable m_0 is a boolean, 1 for $\geq$ and 0 for $\leq$.
  For a *DInstSeq* $d_{seq}^{(1)}$ that has directed a *DMDTraveller* T_d, and the previous *DInstSeq* $d_{seq}^{(0)}$ that has directed T_d, calculate their sequential Euclidean distances as f_1 and f_0. Then algorithm determines if $\frac{f_1}{f_0}$ m_0 m_1.

- *BOOLEAN_DELTA_MEASURE_ALONG_AXIS*

  [-] conditional arguments: $d\_b\_f_0$; $d \in \{l, r, t, b\}$, $b \in \{0, 1\}$,$f_0 \in [0, 1]$

  [-] example: t_1_0.4

  [-] For a *DInstSeq* d_{seq} with point sequence $\overrightarrow{P}$, run $\overrightarrow{P}_0 \leftarrow NOJAG(\overrightarrow{P}, d)$. Then calculate
  $$r = 1 - |\overrightarrow{P}_0|/|\overrightarrow{P}|.$$
  If $b = 1$, then determine if $b \geq f_0$, otherwise if $b \leq f_0$.

If the condition outputs "False", then *DelineCrypto* does nothing. Otherwise, the lines after the conditional arguments specify the instructions to take.

TAKE REACTION AS
"reaction_type"
"reaction_statement"
ENDREACT
"comment: optional below (if character instructions required as targets for reaction)"
ON
"targets_separated_by_comma::(reaction_targets)"

Remark. The format remains the same, that is, the characters in quotes are variables. Additionally, the line that starts with "comment:" is to be omitted for the actual construction of the string representation of any *RInst* instance in the *RInst* file.

There are two categories of reactions, "MODUTG" and "MODINSTR". The "MODUTG" reactions provide instructions for a *UTGSwapper* to conduct swaps. Each "MODINSTR" reaction modifies at least one *DInstSeq* instruction.

The format for "MODUTG" is

MODUTG
"swap_type","number of iterations","optional< *AbstractPRNG* `string` >",
"optional< *bool* >"

The argument "swap_type" is one of *RSwap*, *ObjSwap*, and *TNSwap*. If the "swap_type" is *RSwap*, then there are no two optional arguments. The "optional< *bool* >" is used solely by *ObjSwap* to specify if it should consider swapping the highest-scoring swap (argument set to "True") or the lowest-scoring swap (argument set to "False"). If either *ObjSwap* or *TNSwap* is set, then the *AbstractPRNG* string argument is non-empty and used for decision-making. *ObjSwap* uses it for tie-breakers, in the case in which it must consider more than one highest or lowest scoring swap. *TNSwap* uses it for considering what token to "target" (conduct swaps to minimize its distance to its wanted placement), and also for tie-breakers on highest-scoring swaps involving that target token.

The "MODINSTR" additionally uses the two optional lines, "ON" "targets".

MODINSTR
"*AbstractPRNG* string"
ON
"c,h,a,r,t,e,r,_,s,q"

The "targets" line is a comma-separated sequence of characters that are usually unique denoting the *DInstSeq* instances to be modified by "MODINSTR". Recall in the ending remark of Section 4.2 that each *DInstSeq* corresponds to a character during *DelineCrypto*'s message encoding process. The "*AbstractPRNG* string" is a decider for the *RInst* of category "MODINSTR" on some *DInstSeq* d_{seq}, enabling the *RInst* instance to "react" on d_{seq} by these changes.

- PERM: permutation, permutes the ordering of the *DInst* instances in d_{seq}.
- MODIFY: modifies a *DInst* instance of d_{seq}, preserving its category ("MO","BG", "HOP", and "MSK") while changing its arguments for that category.
- DUP: duplicates a *DInst* instance in d_{seq} and inserting the duplicate at an index of d_{seq} specified by the command's *AbstractPRNG*.
- INS: inserts a new *DInst* instance, constructed using the command's *AbstractPRNG*, at an index specified by that same *AbstractPRNG*.
- DEL: deletes a *DInst* instance of d_{seq} at an index specified by the command's *AbstractPRNG*.

Remark. The number of changes, each of one of the categories that have been listed, has not been constrained.

5 System Encryption

An overview of the components in *DelineCrypto*'s centerpiece, the *DCBot*, is necessary before moving on to the encryption processes. The *DCBot* is a bot, capable of autonomous activity using the *RInst* and *DInst* files provided. The letters "DC" stand for *DelineCrypto*. *DCBot* has two components, another bot called *DBot* that is responsible for operating a *DMDTraveller* instance to produce encodings for a message and a structure called a *Reactor* that uses the *RInst* file to "react" with the structures found in the *DInst* file.

5.1 Instantiating a *DCBot*

In order for a user to instantiate a *DCBot* structure, these following variables must be known and valid at declaration.

- *ifp*: filepath for *DInst* instructions.
- *rfp*: filepath for *RInst* instructions.
- *dfp*: filepath for n-dimensional dataset for use with a *DMDTraveller* instance.
- *difp*: filepath containing instructions for *DMDTraveller* on dataset specified by "dfp" that provides it with a pattern on delineating each two-dimensional dataset (by *NOJAG*, *NODUP*, or *NOCROSS*, and by clockwiseness).
- *utgfp*: a pair of values,

  [0]a boolean specifying if the second argument contains a filepath

  [1]a pair of values; the first value is an *AbstractPRNG* string used by the *UTGSwapper* for decisions (primarily as a tie-breaker), and the second value is a filepath (if element 0 is "True") or a string used to generate a permutation graph.
- *character_sequence*: a vector of characters constituting the entire alphabet of a message (must be a superset to any message that is to be encoded).
- *inputf*: filepath to a file containing the message to be encoded.
- *outputf*: a pair of two filepaths,

  [0]filepath for the encoded message, consisting of a sequence of comma-separated real numbers.

  [1]filepath for a sequence of integers.

5.2 Setting Limits For the Encrypting Components

As of the time of this writing (August 2023), there have not been hard limits imposed on the *DInstSeq* instructions or on the *RInst* reactions. But it is strongly suggested that hard limits be implemented because of the numerically intensive nature of the *DelineCrypto* system.

The rest of this section 5.2 covers the proposed limits for the *RInst* and *DInst* files.

Variable	Limit
number of iterations	$[1, 25]$
length of direction/scale vector	$[2, 20]$
elements of scale vector	$[-2.0, 2.0]$

Table 1: Constraints for some of "MO" command's variables.

Variable	Limit
number of iterations	$[1, 25]$
min/max range	$[-128, 128]$

Table 2: Constraints for some of "BG" command's variables.

Mask Type	Numerical Sequence Length	Numerical Range
FOX	16	$[1, 100]$
SNAKE	14	$[1, 100]$
HAWK	6	$[1, 100]$

Table 3: Constraints for some of "MSK" command's masks. The numerical sequence for a mask command corresponds to ≥ 2 linear congruential generators.

At the beginning when a *DCBot* has been instantiated, each *DInstSeq* has a total number of commands in the range $[4, 12]$ and must have at least two "MO" commands. Each *RInst* command can produce a number of changes in the range $[3, 20]$, each change one of the categories specified in the section on *RInst*s (4.3). During the course of messaging, the number of commands a *DInstSeq* can fall out of the range of $[4, 12]$, given the condition that it has at least two "MO" commands.

The section on *AbstractPRNG*s mentions that more types could be added, and that the list of *AbstractPRNG*s would greatly benefit from more powerful types. In general, the *AbstractPRNG* string format is

`type n u m b e r s.`

The first argument specifies the type of *AbstractPRNG* and the second argument is a sequence of space-separated numbers, each in the range $[0, 1023]$.

5.3 Less Work with Structure Generators

There are two main categories of generators in *DelineCrypto* outside of pseudo-random number generators. The *KeyGen* structure generates *RInst* and *DInst* files. And the *UGraphGen* generates a permutation graph. Both of these generators depend on at least one *AbstractPRNG* instance, and the code for them is quite modifiable by a knowledgeable programmer. These generators' dependencies on *AbstractPRNG*s cannot be guaranteed to produce secure structures (the *RInst* and*DInst* instructions, and the permutation graph).

5.4 The Encryption Procedure

The encryption procedure starts after a *DCBot* instance has been instantiated. The *UTGSwapper* instance created by the *DCBot* has the set of tokens and the set of vertices equal to the set of characters in the "character_sequence" variable. For each index i of the "character_sequence" C_{seq}, the numerical value of the character "character_sequence[i]" is i.

$$NUM_{char} : c \rightarrow C_{seq}.index\_of(c).$$

The *DCBot BOT* iterates through the contents of the message contained in "inputf", character by character. For each character c, *BOT* maps the character to its appropriate *DInstSeq* d_{seq},

$$C_{map} : c \rightarrow G.P(c),$$

found in "ifp", and then produces a two-dimensional point sequence $\overrightarrow{P}$ using d_{seq}. It then writes down the comma-separated flattened version (one-dimensional) of $\overrightarrow{P}$ into "outputf[0]". *BOT* also writes down the numerical value of c according to this mapping,

$$VALUE : c \rightarrow NUM_{char}(C_{map}(c)),$$

into "outputf[1]". Then it iterates through the sequence of *RInst* instances. The only relevant *RInst* instances for character "c" are the "FOR" types of *RInst* if character "c" matches them, and all of the "ANYTIME" types. For each relevant *RInst* instruction, if its condition outputs "True", *BOT* has the *RInst* instance react appropriately, either on a sequence of target characters or on the permutation graph.

6 System Decryption

The following components are necessary for a receiver to decrypt a message.

1. The corresponding *DInst* file.
2. The corresponding *RInst* file.

3. The data file used for delineation.
4. The instructions for a *DMDTraveller*.
5. The file representing the corresponding permutation graph.
6. The character sequence.
7. The sequence of integers representing character enumerations of the encrypted message.
8. The point sequence travelled by the *DMDTraveller* to produce the private cipher corresponding to the encrypted message.

This list of items contains identical values to variables required to instantiate the *DCBot* that performed the encryption. *DelineCrypto*'s encoding scheme follows the template of secret key cryptography. Item 7 is the ciphertext that should be transmitted only in the secure messaging stream. The private key that should be known by both the encoder and decoder consists of items 1-6. Additionally, the sender has item 8, the point sequence $\overrightarrow{P}$.

Items 1-6 are each used by two agents to instantiate *DCBot* instances. All *DCBot* instances with identical arguments (items 2-6) must be synchronized at any time of operation (encryption,decryption). This is because changes occur on *DInstSeq* instructions and the permutation graph by the *RInst* instructions. An important aspect of decryption is that a receiver of a message must take virtually the same processing of information via the *DMDTraveller* as for encryption. The decryption process of *DelineCrypto* can thus be described as replicative of the encryption process.

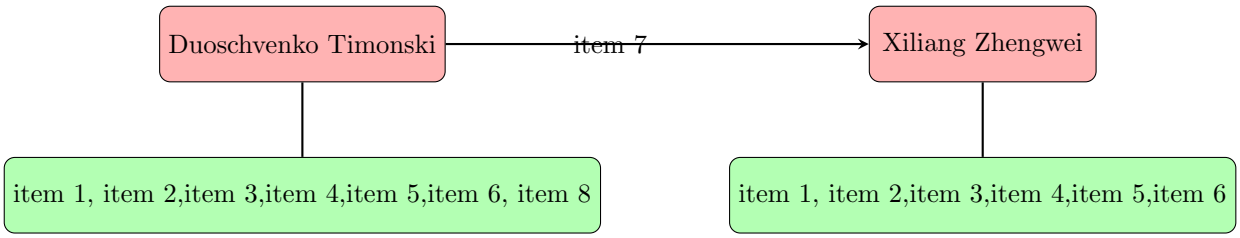

Figure 9: Red boxes are messaging agents, each green box contains items pointed to their red box, and the ciphertext is sent through a secure channel to Xiliang Zhengwei from Duoschvenko Timonski.

The setting where *DelineCrypto*'s communications take place is colloquially referred to as a *messaging stream*.

Definition 6.1 (Messaging stream (in *DelineCrypto*))**.** A secured network W of M, $M \geq 2$ members, in which all members of the stream have the following:

- identical *DCBot* instances
- access to all sent messages in W.

Figure 9 is a visual description of a *messaging stream* between two agents that are using the same *DCBot* to communicate. The decryption procedure is this series of steps.

1. Both Duoschvenko Timonski and Xiliang Zhengwei instantiate their *DelineCrypto* programs to the same *DCBot*s.
2. Duoschvenko Timonski has message X to send to Xiliang Zhengwei, so he has his *DCBot* encrypt X. After X has been encrypted to Y_0 (the ciphertext as an integer sequence) and Y_1 (the private ciphertext), *DCBot* secures the Y_1.
3. Duoschvenko Timonski passes Y_0 through the secure messaging stream to Xiliang Zhengwei.
4. Xiliang's *DCBot* iterates through each integer i in Y_0 and does the following:

   [-] Get the character c corresponding to i by

   $$VALUE^{-1}(i).$$

   [-] Run the encryption process for c to update its *DCBot* component variables that might have been changed.

   [-] Concatenate c to *decrypted_msg*.

   [-] Send P', the point sequence corresponding to the *DInstSeq* for c, to Duoschvenko Timonski's *DCBot*.

   [-] Duoschvenko Timonski's *DCBot* checks P' against the relevant subvector of its own point sequence (item 8). For example, if $\overrightarrow{P}'$ starts at index 100 of item 8, and is length 42, then Duoschvenko Timonski's *DCBot* checks its point sequence at the appropriate range for the answer to

   $$\overrightarrow{P}' \stackrel{?}{=} \overrightarrow{P}[100 : 100 + 42].$$

   If the answer is "False", then Duoschvenko Timonski's *DCBot* assumes that communication with Xiliang Zhengwei has been compromised, and ceases all communication with her.

One major detail a reader might note is that the ciphertext, that consists of an integer sequence, is of equal length to the message that has been encrypted. This detail poses a security risk to the messaging stream, and will be discussed in the next section.

Another detail about the decryption process is that there is "real-time" security between parties in a messaging stream. Note that the *DCBot* component variables are inevitably changed through *RInst* reactions, given the condition that the *RInst* instructions have variance that passes standard statistical metrics. The integer sequence ciphertext passed from Duoschvenko Timonski is deciphered one integer at a time by Xiliang Zhengwei. When Xiliang Zhengwei deciphers each integer to a character, she replies to Duoschvenko Timonski with some integer sequence $\overrightarrow{P}'$. If at any character during the decryption process of a message, Xiliang Zhengwei replies to Duoschvenko Timonski with an incorrect $\overrightarrow{P}'$, Duoschvenko Timonski can assume the messaging stream has been compromised. Otherwise, after Xiliang Zhengwei decrypts the message, her *DCBot* configuration of values will be identical to that of Duoschvenko Timonski. Their *DCBots* are synchronized to a configuration that the adversary will have a difficult time guessing without additional information.

The decryption process described constitutes a kind of "handshake security" that allows Duoshvenko Timonski to know that Xiliang Zhengwei has the correct private key. If Xiliang Zhengwei has an incorrect private key, then she will have a low likelihood of replicating the integer sub-sequence $\overrightarrow{P}'$ that results in a high likelihood of failing this verification test by Duoschvbenko Timonski's *DCBot*. An appropriate term for this protocol is **security through real-time correspondence**, and was inspired by the CAPTCHA tests popularized by organizations such as Google.[BMrc]

7 Possible Attack Vectors on *DelineCrypto*

This section briefs some ways an adversary can "crack" the encryption of a messaging stream between two messengers with the same *DCBot*. At the most secure phase of defense from an adversary, the only element known about a messaging stream is the ciphertext, the integer sequence. Recall that the integers mapped from each character in the original message depends on the the $VALUE$ function in Section 5. The most popular way to "crack" an integer sequence would be to employ attacks using frequency analysis.

The *DelineCrypto* system has the additional feature of *security through real-time correspondence*. By requiring that a receiver of a message in a messaging stream respond to the sender with a point sequence for each character it decrypts in the ciphertext, the sender is granted the knowledge of knowing if the receiver is authentic at any point in time during communication. *DelineCrypto*'s complex anatomy spearheaded into its *DCBot* results in point sequences generated from a message to have the difficulty of replication proportional to the dataset D used for encryption and the *DInst* instructions commanded to the *DMDTraveller*.

7.1 "Cracking" the Ciphertext

An adversary's first target of attack in a messaging stream of *DelineCrypto* is the integer sequence ciphertext, and they will use attacks based on frequency

analysis.

Definition 7.1 (Frequency analysis attack)**.** Given a ciphertext-text $\overrightarrow{N}$, an adversary A that intends to decipher $\overrightarrow{N}$ to the original message $\overrightarrow{C}$, formulates a hypothesis on the natural language L of $\overrightarrow{C}$, and conducts statistical tests on $\overrightarrow{N}$ to determine correspondences between character frequencies and arrangements of L with $\overrightarrow{N}$. This correspondence is then used to produce the hypothesis of $\overrightarrow{N}$ as $\overrightarrow{C}' \approx \overrightarrow{C}$.[Fre]

Frequency analysis attacks do not work, at least efficiently and seamlessly, against standard encryption algorithms such as Advanced Encryption Standard (AES). But the proposed encryption scheme that *DelineCrypto* uses is prone to this type of attack. Here are some examples to aid in understanding this vulnerability.

1. $$ENCRYPT(\texttt{tree}) \rightarrow 4, 5, 9, 9$$

   [-]Adversary notices that there are not many four letter words with two ending letters that are identical. Adversary uses frequency analysis alongside a standard English dictionary to decyprt "4,5,9,9" as "tree".

2. $$ENCRYPT(\texttt{i love tree}) \rightarrow 37, 20, 8, 11, 10, 9, 20, 4, 5, 9, 9$$

   [-]The same adversary in example 1 decrypts the last four integers as "tree". Then they correctly guess that "20" is a space character. They correctly guess "37" as "I". Then they use frequency analysis alongside a standard English dictionary to produce four-letter words that end with "e" and also do not share any letter with "tree" except for "e". The possible candidates such as "hate" are eliminated, leaving the adversary with the best guess of "love".

3. $$ENCRYPT(\texttt{stepped on}) \rightarrow 1, 2, 3, 4, 4, 3, 6, 5, 7, 9$$

   [-]Using frequency analysis alongside an English dictionary, an adversary correctly guesses "1,2,3,4,4,3,6" as "stepped", and then correctly deduces "5,7,9" as " on".

The three examples described use adversaries that make an important assumption about the ciphertext. The assumption is that the permutation graph of the pertinent *DCBot* does not conduct any token-swapping. **Indeed, if no token-swapping were to occur, then frequency analysis would be fun and so much easier for an adversary!**

Suppose that reactions do occur at a relatively high frequency $q \in [0, 1]$ between two agents in a messaging stream. If $q \geq 0.5$, and the adversary knows q, then they would be forced to first calculate an additional mapping to get from

the original ciphertext to the ciphertext if there had not been any token-swaps, and then conduct frequency analysis attacks.

The upper bound of guessing a *DelineCrypto*'s arbitrary ciphertext $\overrightarrow{C}$ for a message $\overrightarrow{M}$ belonging to a character set S by brute-force is $O(|S|^{|\overrightarrow{C}|})$. This is non-polynomial time, which would be problematic for the adversary, and without additional context for them, they could not confidently eliminate credible candidates to know $\overrightarrow{M}$. For example, what is this three-word statement?

```
I am ?cool?dumb?ugly?wild?.
```

Remark. The standard English language has a character size of 96, although the usage of these characters leans heavily towards alphabetical characters, especially vowels. Other natural languages that are not derived from Latin such as Mandarin (via Pinyin) use similar character-based inputs to form their words.

But the upper bound $O(|S|^{|\overrightarrow{C}|})$ is a gross estimate and assumes an adversary does not have any idea what the character identities of a particular integer or set of integers are over their multiple occurrences throughout a message. If no token swaps occur on the permutation graph, a skilled adversary would take a fraction of $O(|S|!)$ attempts to discover the $VALUE$ function by frequency analysis. **The safe assumption for a user is that the adversary, after a prolonged attack session that grants it contextual information on the user's communications, will be able to at least partially decrypt some messages.**

Other types of attacks that have application to modern cryptographic systems are *linear* and *differential* cryptanalysis. [Hey17] **This paper does not cover scenarios where an adversary employs such advanced techniques in their arsenal.**

The safe assumption for agents in a messaging stream is that the adversary does not know the permutation graph. But if the adversary correctly guesses the language L in use between the agents, then they can correctly guess the number of vertices while the question remains of the specific edges. If the *RInst* used produces a high frequency of reactions, and there are a lot of swaps, the adversary will be able to hypothesize swapping routes and vertex identities over an extended amount of time that a particular permutation graph is utilized. The basic assumption an adversary has,if they know the number of vertices, is that the permutation graph is a complete graph.

Definition 7.2 (Simple undirected complete graph)**.** A classification for a simple undirected graph G that satisfies this condition: for every pair of vertices $u, v \in G.V$ such that $u \neq v$, $\{u, v\} \in G.E$.

The assumption that the permutation graph is complete means that the adversary will have a larger set of legitimate guesses for each swap conducted on the permutation graph. An adversary, in their attempts to decipher the encrypted message, must hypothesize that swaps occurred at particular indices in the encrypted message. Swaps change the integer value of a character c at

index i in the ciphertext $\overrightarrow{N}$, that otherwise would have an identical integer to c in the previous parts of the ciphertext, $\overrightarrow{N}[:i]$. To illustrate, suppose the permutation graph is a path. Then the swap of two tokens t_u, t_v respectively on neighboring vertices u, v is easier for the adversary to discover than if they had to deal with graphs with higher degrees of connectivity. In practice, paths between vertices of a complete graph are uniform in length and thus requires fewer swaps to solve a token. **So *RInst* instructions that not only produce a high number of swaps but also result in contiguously non-identical token placements of a permutation graph over time are strongly advised.**

Note that this is only one layer of encryption on the original message that relies on the organic mechanisms of *DCBOT*s prone to unpredictable changes in behavior (expressed via the ciphertext pairs) in order to preserve secrecy of communications. Certainly, more layers could be added on top of this first layer.

7.2 "Cracking" Security Through Real-Time Correspondence

Suppose an adversary A were to obtain a sequence of ciphertexts $\overrightarrow{C}$ in a messaging stream between two agents. Furthermore, after A conducted extensive analysis that used contextual information on the two agents with regards to their communications X, A has produced a hypothesis H on

$$DECRYPT(\overrightarrow{C})$$

that they think is valid with passing confidence $T_0 \geq T$ (T a minumum threshold).

But how would A have been able to access the messaging stream if it was secured? Perhaps A used a technicality and now has unchallenged access to the messaging stream. A can read all ciphertexts transmitted across the stream between both agents. This type of attack exemplifies the man-in-the-middle attack.[Mitb] [Mita] This scenario has already been considered, although not explicitly stated in these terms, in Section 7.1.

If A is able to somehow gain administrative access to the receiver, and additionally is able to maintain knowledge of the verification point sequences $\overrightarrow{P}'$ that need to be replied back to the sender, then **security through real-time correspondence** has been compromised.

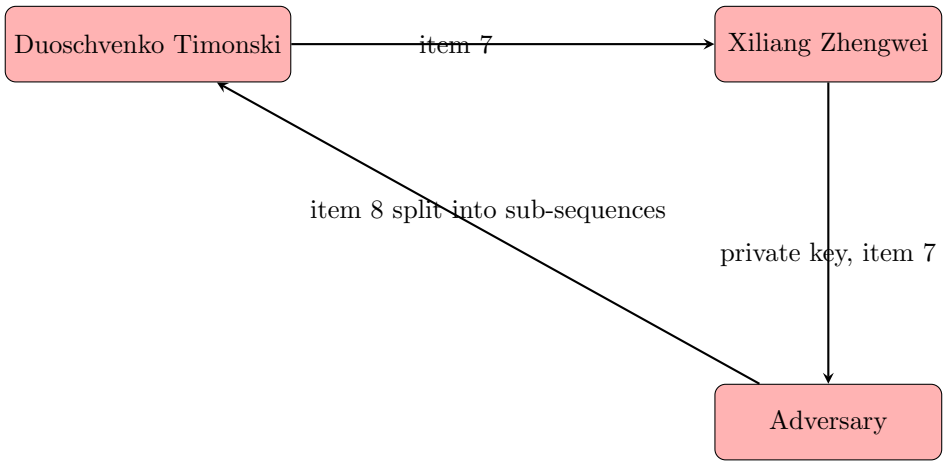

Figure 10: Communication from Figure 9, but with a man-in-the-middle attack by Adversary. Adversary is able to read all ciphertexts from Duoschvenko Timonski to Xiliang Zhengwei, and Xiliang Zhengwei's *DCBot* relays the point sequence $\overrightarrow{P}'$ to the impersonator Adversary, who then transmits $\overrightarrow{P}'$ to Duoschvenko Timonski's *DCBot*. Duoschvenko Timonski remains clueless about Adversary's impersonation during this time.

Security through real-time correspondence makes two important asumptions.

- All members of a messaging stream are fully aware of each other's existence in the stream.
- All members on the receiving end of any arbitrary message are fully capable of replicating the point sequence $\overrightarrow{P}$ that corresponds to the ciphertext.

It is obvious that if an adversary were to gain administrative access to the receiving end, then they could "take over" the correspondence, allowing them to maintain trust with the sender so that the sender will not cease communications due to suspected compromise of the messaging stream.

If A were to have a lower privilege in a man-in-the-middle attack so that they do not know the private key, then A is stuck at formulating their own decryption algorithms for the ciphertexts they are able to obtain. A secure implementation of *DelineCrypto* means A will not be able to alter communication in the channel or impersonate sender or receiver. By the first assumption of security through real-time correspondence, members of the sending end of the messaging stream are fully aware of A's existence and that A is an adversary because they cannot reply back with a correct point sequence $\overrightarrow{P}'$. To achieve an "accepted" status in the messaging stream, A would then be forced to guess the private key.

8 The End

DelineCrypto is a pseudo-autonomous system that relies on permutative ciphers to secure the secrecy of messages. **The unique quality of *DelineCrypto* rests not on this style of cipher but on the mechanisms behind its pseudo-autonomous activity.** Its centerpiece, the *DCBot*, relies on complex instructions that allow it to alter the token placement of a permutation graph and to alter those instructions themselves. The *DCBot*'s two metaphorical arms are its permutation graph and its security agent *DMDTraveller* in charge of generating the private ciphertext.

Pseudo-random number generators are an important factor in preserving the secrecy of the private key, especially the instructions of the *DInst* and *RInst* files. The "reactive" patterns of instructions given to the *DMDTraveller* allow the structure to not fall into patterns of movements, outputted as point sequences, that can be reasonably predicted by an adversary over the adversary's course of attack. Note that this system is still under development, and many design quirks have not yet been finalized. This paper is intended to give the reader a glimpse of the fundamentals behind this system.

Sources

[MT02] George Marsaglia and Wai Wan Tsang. "Some difficult-to-pass tests of randomness". In: *Journal of Statistical Software* (2002).

[Hey17] Howard Heys. "A Tutorial on Linear and Differential Cryptanalysis". PhD thesis. Memorial University of Newfoundland, 2017.

[BMrc] Prakash S. Bodkhe and Dr. P. N. Mulkalwar. "CAPTCHA Techniques: An Overview". In: *International Journal on Recent and Innovation Trends in Computing and Communication 5(9):15-21* (March 2021).

[al.a] Erik Demaine et al. *Lecture 9: Breadth-First Search*. URL: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/196a95604877d326c6586e60477b59d4_MIT6_006S20_lec9.pdf. (accessed: 07.22.2023).

[al.b] Jun Kawahara et al. *The Time Complexity of Permutation Routing via Matching, Token Swapping and a Variant*. URL: https://arxiv.org/pdf/1612.02948v2.pdf. (accessed: 7.25.2023).

[Cha] Austin Chamberlain. *Applications of Cryptography*. URL: https://blogs.ucl.ac.uk/infosec/2017/03/12/applications-of-cryptography/. (accessed: 07.22.2023).

[Fre] *Cryptanalysis*. URL: https://owasp.org/www-community/attacks/Cryptanalysis. (accessed: 8.8.2023).

[Nis] *Cryptography*. URL: https://www.nist.gov/cryptography. (accessed: 07.22.2023).

[Staa] *Find all paths between two graph nodes*. URL: https://stackoverflow.com/questions/9535819/find-all-paths-between-two-graph-nodes. (accessed: 7.28.2023).

[Mita] *Man-in-the-Middle Attack: Types and Examples*. URL: https://www.fortinet.com/resources/cyberglossary/man-in-the-middle-attack. (accessed: 8.9.2023).

[Sha] Cosma Shalizi. *Lecture 17: Multi-Collinearity*. URL: https://www.stat.cmu.edu/~cshalizi/mreg/15/lectures/lecture-17.pdf. (accessed: 7.24.2023).

[Stab] *The proof that Finding all simple paths between two nodes in an undirected graph is NP*. URL: https://stackoverflow.com/questions/64051778/the-proof-that-finding-all-simple-paths-between-two-nodes-in-an-undirected-graph. (accessed: 7.28.2023).

[Ull] Jeffrey D. Ullman. *Chapter 11: Dimensionality Reduction*. URL: http://infolab.stanford.edu/~ullman/mmds/ch11.pdf. (accessed: 7.25.2023).

[Mitb] *What is a man-in-the-middle attack?* URL: https://us.norton.com/blog/wifi/what-is-a-man-in-the-middle-attack. (accessed: 8.9.2023).