A Complete 3BLD Tracing Algorithm for Pure Letter Memorization Using Breaks

1 Introduction

I'll start this by saying I do not actually know how to solve a Rubik's cube blindfolded (plus I'm barely sub30 on the 3x3), but I've been working towards learning it. I've found that the cubing community really seems to favor video tutorials, but it doesn't feel like too many of the resources are spread out across google docs and videos. This document is my attempt at consolidating the memorization portion of 3BLD into one document that covers all possible cases.

From my understanding, twists and flips are typically memorized separately, as flip algorithms typically aren't too difficult, and advanced 3BLD solvers even use LTCT and/or other 2-3 twisted/flipped corner/edge algs. However, for several reasons I use the breaking method for flips and twists:

1. Memorization only contains letter pairs (Possible benefit for MBLD)
2. Seems like less algorithms will need to be learned (I could be wrong)

The downside of this method is that tracing becomes marginally more difficult, but I can't really comment on the difficult at the moment, since I still haven't learned 3BLD yet. I'm also aware that Full Floating is an advanced technique to handle new cycles, but for the same reason I'm using breaking into twists/flips (memorization is more homogeneous, and less algorithms), I'll stick with traditional cycle breaking.

2 Basic 3BLD method

Those familiar with blind solving can skip this part. 3BLD is typically performed via algorithms that permute a few pieces, leaving other pieces unaffected. As such, solves are typically separated into solving corners and solving edges.

To know what pieces need to be permuted, each piece is labeled. It would seem that in the past numbers used to be used, but nowadays, letters seem to be the dominant strategy, with the most common lettering scheme being the Speffz lettering scheme, which is shown below (different lettering schemes do exist and aren't uncommon).

```
    AAB
    D B
    DCC
EEF IIJ MMN QQR
H F L J P N T R
HGG LKK POO TSS
    UUV
    X V
    XWW
```

2.1 Old Pochmann (OP)

The widely accepted entry level method for blind solving is the Old Pochmann method, which recycles PLL algorithms from the CFOP method to perform the permutations. The T Perm is used to swap edges B and D, and the Y Perm is used to swap corners A and P. Conjugates (an algorithm of the form ABA' denoted A:B that sets up to a permutation, does the permutation, and undos the setup) are used to bring pieces to a swapping location, to swap with a buffer piece (B for edges, A for corners).

Those familiar with PLL will know T and Y perm will also swap another pair of pieces in addition to the two previously mentioned (T swaps corners B and C, Y swaps edges A and D). However, if there are an even number of permutations, those instances will cancel out. If an odd number of permutations needs to occur for one, a parity algorithm would need to occur that swaps both of those piece pairs (for math reasons I might write up later, an odd number of permutations occurs in edges iff the number of corner permutations is also odd). Luckily, those swaps are covered by the Ra perm, which would just need to be performed between edge and corner execution.

For actual execution, solvers perform what is known as a cycle sort. Cycle sort is what is known as an

```
function CycleSort($a = [a_1, \dots, a_n]$)
    for $i$ in $[1..n-1]$ do
        while a[i] is not in correct position do
            $k \leftarrow$ correct position of $a[i]$
            Swap $a[i]$ and $a[k]$
```

in-place sorting algorithm that aims to minimize the number of write (or swap) operations[1]. This is good for 3BLD for several reasons:

1. Our only operations are permutations, so our algorithm would need to necessarily be in-place
2. Minimizing swaps is the same as minimizing algorithms for execution, which is nice
3. Each element is read exactly once, which makes memorization faster
4. Elements are swapped in roughly the same order they are read (with the exception of when the While condition is satisfied, in which case no swap occurs for that read). This makes memorization especially easy as execution occurs in the same order you memorize the pieces.

Adapting this to the cube, we get the algorithm shown on the next page. Here we simply define cube as a mapping from a position string to a color string (e.g. UFR maps to WGR in a white top green front solved cube), and the lettering scheme as an invertible from letter to position string, POS, and from letter to color string, COL.

While at a glance the algorithm looks quite a bit longer than the 4 lines in cycle sort, this method is basically doing the exact same thing, there's just a little more overhead to separate the list access and swaps into two different processes. Reading the comments left on the right side of the algorithm should give you the rough gist of how 3BLD solving is performed, and you can look at the algorithm itself for the details.

Regular blind solvers (and people who read the introduction) probably noticed that something is missing from the algorithm I provided. Up to this point, I haven't addressed corner twists and flipped edges yet. This is because there are several ways to address those, and the method I'll introduce a in a bit is a nonstandard method. For more details on the more popular ways to handle twists and flips, please refer to other resources for now.

While for the OP method, it should matter what order you memorize and execute corners and edges, the general accepted order for advanced solves is memorize corners, memorize edges, execute edges, and execute corners. The order of memorization and execution is reversed so the last memorized sequence is executed immediately, allowing you to use a more short term memorization strategy for edges and a longer term method for corners.

3 3-style

I'll skip the other intermediate methods (M2 and Orozco) as they have a similar formulation to OP, but with lower move count. Currently, the most advanced method for 3BLD solving is called 3-style. 3-style uses algorithms that permute 3 pieces at once instead of the swapping that occurs in the OP method, meaning that letters in the memorized sequence is executed in pairs rather than one at a time.

The algorithms used in 3-style are typically of the form of conjugates of commutators. Algorithms that just use commutators without the conjugates are called pure commutators.

[1]Though in practice cycle sort is not used too often as knowing the correct position of $a[i]$ is typically not given and would need extra often linear time computation, making cycle sort a quadratic algorithm

```
Require: $cube : \{p* : p \in \{U, D, F, B, L, R\}\} \to \{c* : c \in \{W, Y, G, B, O, R\}\}$
Require: $POS : \{A..X\} \to \{p* : p \in \{U, D, F, B, L, R\}\}$
Require: $COL : \{A..X\} \to \{c* : c \in \{W, Y, G, B, O, R\}\}$

  function TRACE(cube, pieces, start)
      output ← []
      $curr \leftarrow POS$(start)
      pieces.remove($curr$)
      $cycle \leftarrow \emptyset$
      while pieces is not empty do                 ▷ Do the following until you have checked all pieces
          if $cube(curr) \not\approx COL^{-1}(start)$ then          ▷ If the piece you are looking at is the buffer piece
              if cycle then                ▷ If you were in a cycle add the letter of the piece like normal
                  output.append($COL^{-1}(curr)$)
                  pieces.remove($curr$)
                  $cycle \leftarrow \emptyset$
              while pieces is not empty do                                  ▷ Go through unchecked pieces
                  $curr \leftarrow$ pieces[0]
                  if $cube[curr] \approx COL(POS^{-1}(curr))$ then     ▷ If piece is in right position, stop looking
                      break
                  pieces.pop(0)
              if pieces is emtpy then break
              start ← $POS^{-1}(curr)$                               ▷ Set new buffer to previously found position
              output.append(start)
              $cycle \leftarrow curr$
          next ← $COL^{-1}(cube(curr))$            ▷ Get the letter of next piece from color of current piece
          output.append(next)
          curr ← $POS$(next)                        ▷ Look at piece in the position of the just found letter
          pieces.remove(curr)
      return output

  function OP-OP($cube$)
      corner_letters ← trace($cube$, all 8 corner piece positions, $A$)                  ▷ Perform Memorization
      edge_letters ← trace($cube$, all 12 edge piece positions, $B$)
      for $a$ in edge_letters do                                                          ▷ Execute Edges
          setup ← algorithm to move $a$ to $D$ without affecting corners $B$ and $C$
          Execute Conjugate (setup, T Perm, (setup)')
      if lengths of corner_letters and edge_letters are odd then                         ▷ Execute Parity
          Execute Ra Perm
      for $a$ in corner_letters do                                                      ▷ Execute Corners
          setup ← algorithm to move $a$ to $P$ without affecting edges $A$ and $D$
          Execute Conjugate (setup, Y Perm, (setup)')

  function 3STYLE($cube$)
      corner_letters ← trace($cube$, all 8 corner piece positions, $C$)                  ▷ Perform Memorization
      if length of corner_letters is odd then  ▷ Handle memorization side of parity with weak swapping
          Swap $C$ and $D$ entries in $COL$
      edge_letters ← trace($cube$, all 12 edge piece positions, $C$)
      for $a, b$ in edge_letters.paired() do                                              ▷ Execute Edges
          Execute conjugate and commutator algorithm for $(a, b)$
      for $a, b$ in corner_letters.paired() do                                          ▷ Execute Corners
          Execute conjugate and commutator algorithm for $(a, b)$
      if Unpaired letter $a$ at end of corner_letters then                               ▷ Execute Parity
          Execute parity algorithm for $a$
```

3.1 Commutator Conjugates

A commutator is an algorithm of the form (A B A' B'), where you execute one algorithm, then another, then you undo the first algorithm, then undo the other. This is typically notated with square brackets as [A,B]=(A B A' B'). In group theory, a commutator can be thought as a measure of how much two elements commute. For instance, if A=U and B=D, those rotations commute completely, making [U,D] do nothing to the cube. For the case of 3BLD solving, we are interested in the next "smallest" commutator[2] where A and B overlap in exactly one piece, resulting in a cycling of exactly three pieces.

Commutators of sane length only get you so far, so they are often combined with conjugates. Conjugates are algorithms of the form (A B A'), and in group theory it can be thought of performing an operation in a different basis. The idea is that you can perform some moves to setup the desired 3-cycle to a known pure commutator, perform the commutator, then undo the setup so that only those 3 pieces were cycled. A conjugate is often notated with a colon, so (A:B)=(A B A'), and combined with a commutator, (A:[B,C])=(A B C B' C' A').

Excluding the buffer since it isn't memorized, and since a pair never contains letters from the same position, you get 378 possible letter pairs for corners and 440 letter pairs for edges. Since the flip of a letter pair is the reverse of a commutator, i.e. (A:[B,C]) is the inverse of (A:[C,B]), only half of those algorithms are really unique combinations. Most people use intuitive commutators and setups for each algorithm to start, so while it's still a large number, it's far less than it originally seems (though then speed optimized algorithms come into play it may be a different story).

So how do these 3-cycles we get from these commutator conjugates actually work? Well, basically the same as before. When working one letter at a time, we normally swap a piece with a buffer so that piece gets stored in the buffer to eventually go to the location of the next piece, and so on. If instead of swaps we do 3-cycles, we are basically just combining two of the swaps into on step, cycling the piece in the buffer to the first piece, cycling the first piece to the location of the second piece, and moving the second piece into the buffer.

3.2 Parity

As one would imagine parity (cases where on odd number of corners are traced) would need to be handled differently. For this document, I'll use a technique called weak swapping. As you may recall, when there is on odd number of corner letters there must also be an odd number of edge letters, and each letter corresponds to one swap. However, in 3-style only two pieces are cycled at a time. This means that the cases where one corner and one edge remains swapped with the buffer cannot be resolved with a three cycle. One way is to know an algorithm for each edge-corner letter pair for a simultaneous swap like in the OP method, but that requires 462 algorithms.

An alternative easier way to handle parity is what is known as weak swapping. With weak swapping, when we trace edges after noticing parity in corners, we simply map the colors of the buffer to another piece. When executed, this results in that piece being swapped with the buffer, so we technically get an odd number of edge swaps even though we executed an even amount of swaps. From there, we just need 21 algorithms for each corner case. To make the cases easier for those who learned CFOP, the edge piece we swap with the buffer is typically B, setting up the algorithms to be conjugates of Ja, Jb, and Y Perms, making most of the 21 algorithms require only CFOP knowledge and some intuition.

4 Breaking into Twists and Flips

Up to this point, I haven't discussed how to resolve twists and flips. As previously stated, the normal beginner way to perform these is to use a twist/flip algorithm like those learned in the beginner 3x3 method. However, this approach requires you to use a different system to memorize what corners/edges need to be fixed, which felt a bit wrong in my mind.

My ideal method would integrate twists and flips into the memorized letter sequence so we can take advantage of the mnemonics we used to memorize the rest of the execution. As it turns out, there is an

[2] I don't actually know in what measure "smallest" would be, but I assume it would mean in the smallest difference between the start and end state of the cube. I also don't even know if a 3-cycle is in fact the "smallest" commutator in the sense I just described, as a 2 flip can be performed via commutator, and is arguably smaller, so I guess it's most accurate to say a 3-cycle is the smallest position shifting cycle commutator?

existing technique called "Breaking into the twist/flip" that does this, but it's documentation is rather sparse. As far as I know, the main resource for learning this is Timithy Goh[3] and Noah Swor's[4] videos on the subject. However, I, for one, found these to be a bit challenging to follow, and they seemed to omit some details that would lead to a complete explanation.

The gist of breaking into a twist/cycle is to interrupt a letter pair with one letter corresponding to a side on the twisted face, and the letter on the same piece to which the tile from the first letter should be sent towards. One quirk of breaking into a twist/flip is that the second letter in the letter pair you break will change depending on the number of flips you perform. For this reason, I suggest only breaking into the last letter pair to make tracing easier.

To investigate this on my own, I created a quick python simulation to test a complete tracing algorithm that uses these techniques so that the only information transmitted between the memorization and the execution are the letter sequences, while handling any number of twists and flips. The algorithm has been verified on 100,000 random scrambles, solving each one with perfect accuracy.[5]

4.1 Flips

With weak swapping, breaking into a flip is fairly straightforward. The order of the letters when flipping isn't particularly important, so you can choose what's convenient. To resolve how to determine what the last letter should be, you simply need to check the parity of the number of flips you perform. If the number of flips is odd, you need to change the last letter to the letter on the other side of the piece. If the number of flips is even, you can keep the same letter.

4.2 Twists

Corner twists are a bit more complicated than flips. Since there are three possible choices, you can choose the letter pair for the twist that has the easiest algorithm, but for simplicity, I will suggest making the first letter for the twist the one corresponding the the U/D face, and the second letter the tile on the U/D face. Focusing on one set of opposite face pair will likely make subsequent mental calculations easier for handling non-parity cases.

If you don't have parity, you must insert the letter pairs before the last letter in the sequence like when flipping. However, to determine to which letter the last letter should be changed to depends on the sum of the direction of each of the flips. In my opinion, computing this on the fly is a bit challenging, so I came up with a method to compute this with relatively low overhead. For each twisted piece, look at location of the U/D tile on it's face. If it's in the first or third quadrant, add 1, otherwise subtract 1. At the end take the number modulo 3, and that number is the number of times you trace clockwise from the tile the last letter corresponds to.

If you do have parity, you actually can't break the twist letter pair between the last letter pair since that would insert a letter pair from the same tile to be executed as a pair, which is impossible. As such, we can do something easier by just appending the twist letter pairs at the end, making the new parity algorithm correspond to the last letter in the twist sequence. In this case, you don'ot need to make any tally for changing the last letter.

4.3 The rarest of rare edge case

In the off chance your letter sequence is empty meaning that you only have solved pieces and twists/flips, you can put an arbitrary letter twice as the letter sequence as if you were performing a cycle break. The chosen letter must not correspond to the letter of the first or last twist/flip, or else we will have an illegal letter pair in execution.

Putting this all together, we get the algorithm on the next page.

[3] https://www.youtube.com/watch?v=uFPLd4ptUiY

[4] https://www.youtube.com/watch?v=F8ddficCug8

[5] Code can be found at https://github.com/Origamijr/cubing-misc

```
Require: $cube : \{p* : p \in \{U, D, F, B, L, R\}\} \to \{c* : c \in \{W, Y, G, B, O, R\}\}$
Require: $POS : \{A..X\} \to \{p* : p \in \{U, D, F, B, L, R\}\}$
Require: $COL : \{A..X\} \to \{c* : c \in \{W, Y, G, B, O, R\}\}$

  function TRACE(cube, pieces, start)
      output ← []
      $curr \leftarrow POS$(start)
      pieces.remove($curr$)
      $cycle \leftarrow \emptyset$
      while pieces is not empty do                ▷ Do the following until you have checked all pieces
          if $cube(curr) \not\approx COL^{-1}(start)$ then          ▷ If the piece you are looking at is the buffer piece
              if cycle then            ▷ If you were in a cycle add the letter of the piece like normal
                  output.append($COL^{-1}(curr)$)
                  pieces.remove($curr$)
                  $cycle \leftarrow \emptyset$
              while pieces is not empty do                              ▷ Go through unchecked pieces
                  $curr \leftarrow$ pieces[0]
                  if $cube[curr] \approx COL(POS^{-1}(curr))$ then     ▷ If piece is in right position, stop looking
                      break
                  pieces.pop(0)
              if pieces is emtpy then break
              start ← $POS^{-1}(curr)$                         ▷ Set new buffer to previously found position
              output.append(start)
              $cycle \leftarrow curr$
          next ← $COL^{-1}(cube(curr))$             ▷ Get the letter of next piece from color of current piece
          output.append(next)
          curr ← $POS$(next)                       ▷ Look at piece in the position of the just found letter
          pieces.remove(curr)
      return output

  function 3STYLE($cube$)
      corner_letters ← trace($cube$, all 8 corner piece positions, $C$)                ▷ Perform Memorization
      if length of corner_letters is odd then  ▷ Handle memorization side of parity with weak swapping
          Swap $C$ and $D$ entries in $COL$
      edge_letters ← trace($cube$, all 12 edge piece positions, $C$)
      for $a, b$ in edge_letters.paired() do                                          ▷ Execute Edges
          Execute conjugate and commutator algorithm for $(a, b)$
      for $a, b$ in corner_letters.paired() do                                        ▷ Execute Corners
          Execute conjugate and commutator algorithm for $(a, b)$
      if Unpaired letter $a$ at end of corner_letters then                            ▷ Execute Parity
          Execute parity algorithm for $a$
```

5 TODO

- Actually fill out the final algorithm
- Figures
- Examples