Qter: the Human Friendly Rubik's Cube Computer

Arhan Chaudhary, Henry Rovnyak, Asher Gray

Abstract. In this paper/report/whatever, we propose a computer architecture called *Qter* that allows humans to perform computations by manipulating Rubik's Cube by hand. It includes a "machine code" for humans called *Q* and a high-level programming language called *QAT* that compiles to *Q*. The system also applies to other permutation puzzles, such as the 4x4, Pyraminx, or Megaminx. We also present a program we call the *Qter Architecture Solver* that executes on a classical computer to discover Qter architectures on arbitrary puzzles.

https://github.com/ArhanChaudhary/qter/

Our submission to SIGHORSE 2025, the Purdue Hackers journal
https://sig.horse/

Acknowledgments

We extend our sincere thanks to Tomas Rokicki for personally providing us key insight into Rubik's Cube programming techniques throughout the past year. Qter would not have been possible without his guidance. We are immensely grateful for his time.

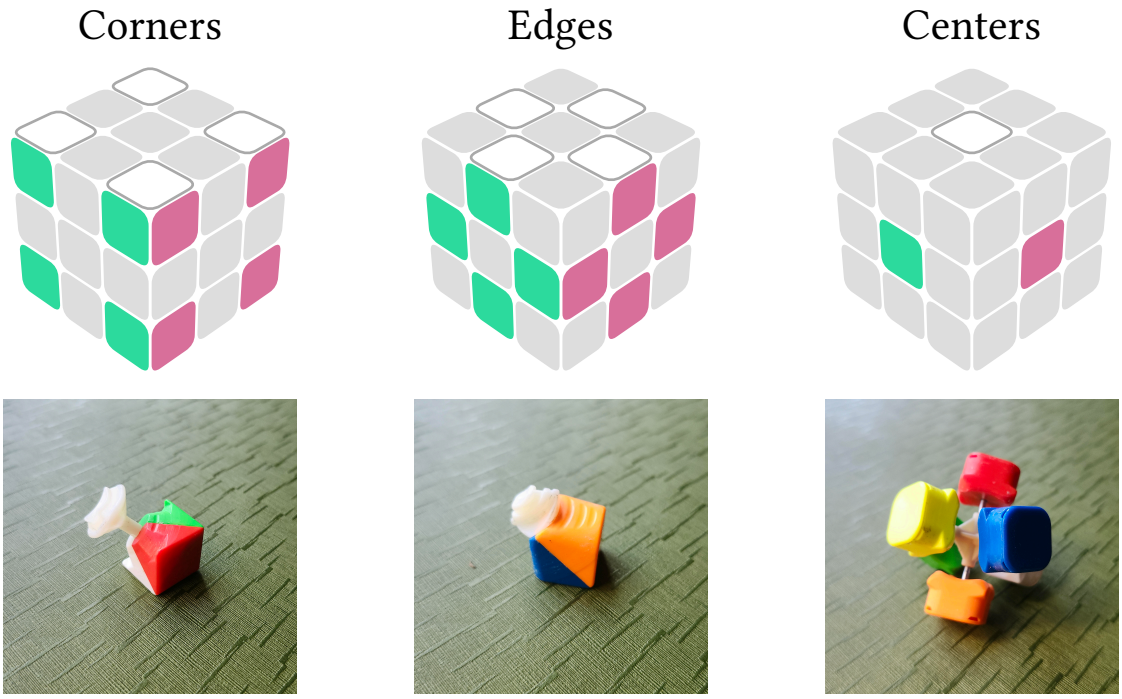
We also extend our gratitude to Ben Whitmore for helping us ideate the initial design of the Qter Architecture Solver.

Contents

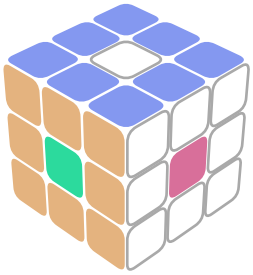
Appendix

1) Background

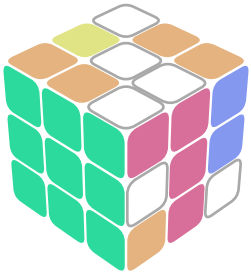
Before we can explain how to turn a Rubik's Cube into a computer, we have to explain what a Rubik's Cube *is* and the fundamental mathematics behind how it works. First, a Rubik's Cube is made out of three kinds of pieces: *Corners*, *Edges*, and *Centers*.



You can see that the centers are attached to each other by the *core* and are only able to rotate in place. This allows us to treat the centers as a fixed reference frame to tell whether or not a sticker is on the correct side. For example, if we have the following scramble,

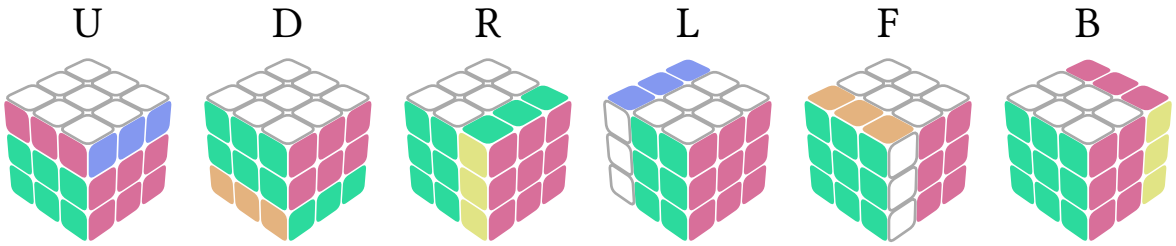


it may look as if the centers are the only thing unsolved, but in fact we would actually consider *everything else* to be unsolved. The reason is that all of the stickers are different from the center on the same side as it. Next, people who are beginners at solving Rubik's Cubes often make the mistake of solving individual stickers instead of whole pieces.

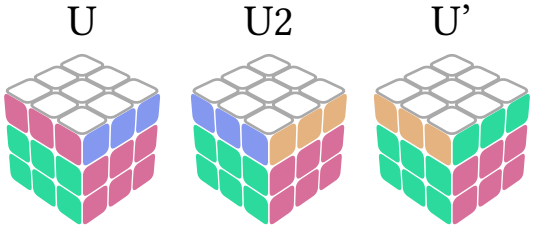


If someone does this, then they haven't actually made progress towards a solution because the stickers on the pieces move together, which means that all of the pieces on the green face in the example given will have to be reshuffled to bring the rest of the stickers to their correct faces. Instead, it's better to solve a full "layer" (3x3x1 block), because all of the pieces are in their correct spots and won't need to be moved for the entire rest of the solve. The takeaway being that in general, *we need to think about the cube in terms of pieces rather than in terms of stickers.*

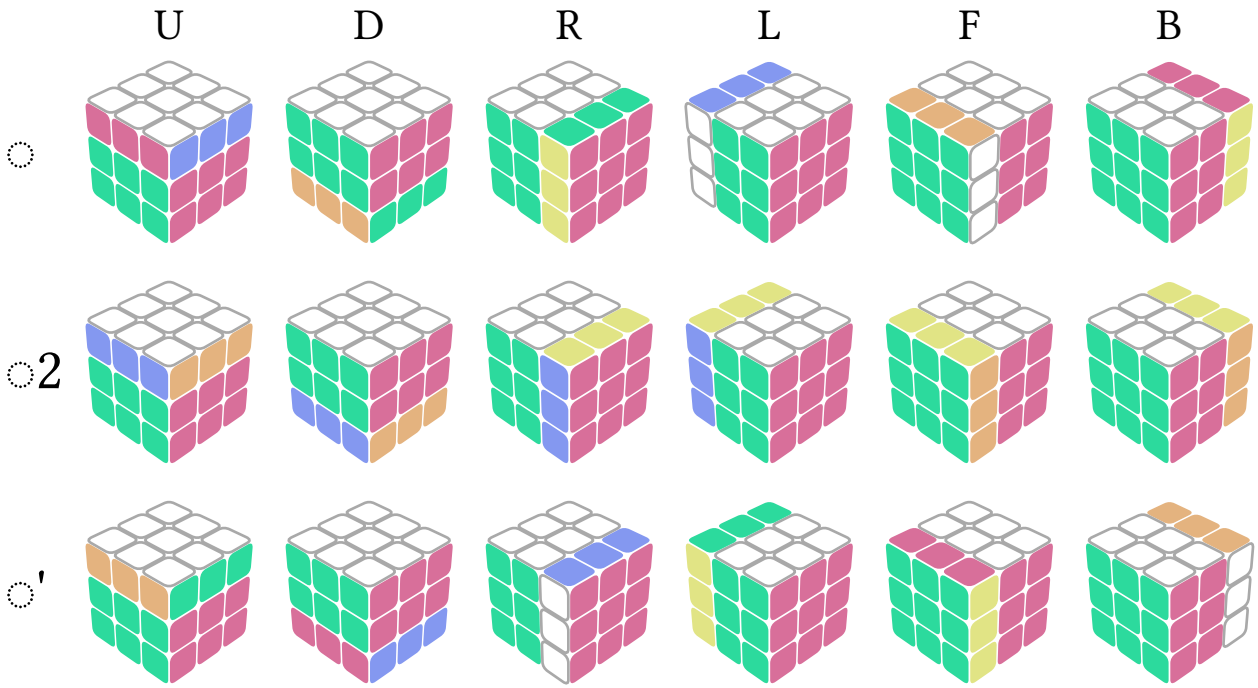
Now, we need some way to notate scrambles and solutions on a Rubik's Cube. We will use the conventional "Singmaster Notation" which is standard in the Rubik's Cube solving community [1]. First, we will name the six sides of a Rubik's Cube *Up* (U), *Down* (D), *Right* (R), *Left* (L), *Front* (F), and *Back* (B). Then, we will let the letter representing each face represent a clockwise turn about that face.



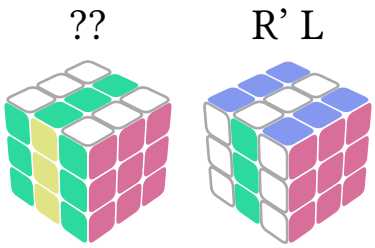
To represent double turns or counterclockwise turns, we append a 2 or a ' respectively to the letter representing the face.



Here is a full table of all 18 moves for reference:

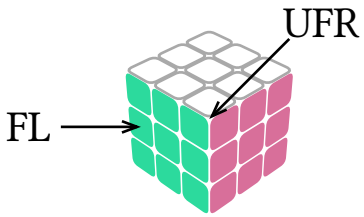


It may look like we're forgetting some moves. After all, there are *three* layers that you can turn, not just two, and we haven't given names to turns of the three middle slices. However, we don't actually need to consider them because "slice" turns can be written in terms of the 18 "face" turns.



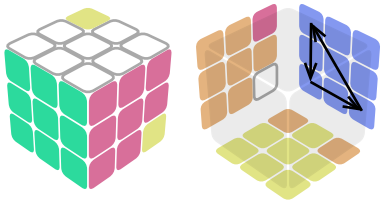
Those two cube states are actually the same because if you take the first cube and rotate it so that the green center is in front and the white center is on top again, we would see that it is exactly the same as the second cube. Since we're using the centers as a reference point, we can consider these two cube states to be exactly the same. Slice turns do have names, but we don't need to care about them for the purpose of this paper.

Another thing that we will need to name are the pieces of a Rubik's Cube. To do this, we can simply list the sides that the piece has stickers on. For example, we can talk about the "Up, Front, Right" or *UFR* corner, or the "Front, Left" — *FL* — edge.



This system is able to uniquely identify all of the pieces. Finally, a sequence of moves to apply to a Rubik's Cube is called an *algorithm*. For example, (L2 D2 L' U' L D2 L' U L') is an algorithm that speed cubers memorize to help them at the very end of a solution when almost every piece is solved. It performs a three-cycle of the UBL, DBL, and DBR corners:

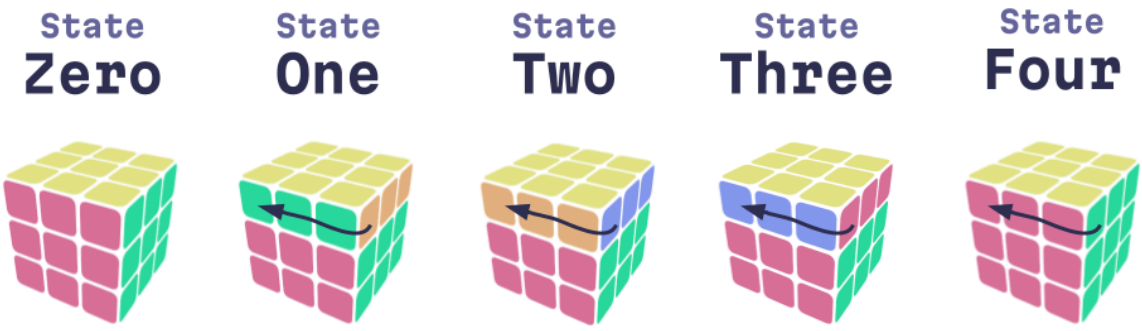
L2 D2 L' U' L D2 L' U L'



2) What is Qter?

Now that you understand what a Rubik's Cube is and the fundamental mechanics, we can explain the ideas of using it to perform computation. The most important thing for a computer to be able to do is represent numbers. Let's take a solved cube and call it "zero".

The fundamental unit of computation in Qter is an *algorithm*, or a sequence of moves to apply to the cube. The fundamental unit of computation on a classical computer is addition, so let's see what happens when we apply the simplest algorithm, just turning the top face, and call it addition by one. What does this buy us?



We can call this new state "one". Since we want the algorithm (U) to represent addition, perhaps applying (U) *again* could transition us from state "one" to state "two", and again to state "three", and again to state "four"?

When we apply (U) the fourth time, we find that it returns back to state "zero". This means that we can't represent every possible number with this scheme. We should have expected that, because the Rubik's Cube has a *finite* number of states whereas there are an *infinite* amount of numbers. This doesn't mean that we can't do math though, we just have to treat numbers as if they "wrap around" at four. This is analogous to the way that analog clocks wrap around after twelve, and this form of math is well-studied under the fancier name "modular arithmetic".

2.0.1) Addition

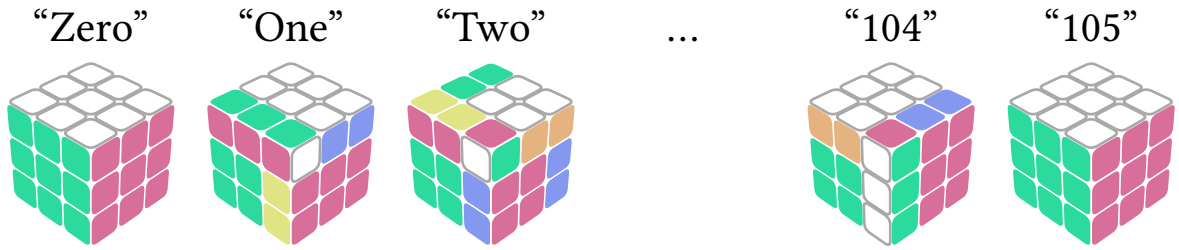
Can we justify this way of representing numbers? Let's consider adding "two" to "one". We reach the "two" state using the algorithm (U U), so if we apply that algorithm to the "one" state, we will find the cube in the same state as if we applied ((U) (U U)), or (U U U), which is exactly how we reach the state "three". It's easy to see that associativity of moves makes addition valid in this scheme. What if we wanted to add "three" to "two"? We would expect a result of "five", but since the numbers wrap around upon reaching four, we would actually expect to reach the state of "one". You can try on your own Rubik's Cube and see that it works.

What if we want to perform subtraction? We know that addition is performed using an algorithm, so can we find an algorithm that adds a negative number? Let's consider the state that represents "one". If we subtract one, we would expect the cube to return to state "zero". The algorithm that brings the cube from state "one" to state "zero" is (U'). This is exactly the *inverse* of our initial (U) algorithm. If we want to subtract two, we can simply subtract one twice as before: (U' U').

You may notice that subtracting one is equivalent to adding three, because (U') is equivalent to (U U U). It may seem like this is a contradiction, but it actually isn't: Adding three to one gives four, but since four wraps around to zero, our result is actually zero, as if we subtracted one. In general, any number can be seen as either positive or negative: $-1 = 3$, $-2 = 2$, and $-3 = 1$. You can manually verify this yourself if you like. Interestingly, this is how signed arithmetic works in a classical computer, but that's irrelevant for our purposes.

2.0.2) Bigger numbers

If the biggest number Qter could represent was three, it would not be an effective tool for computation. Thankfully, the Rubik's Cube has 43 quintillion states, leaving us lots of room to do better than just four. Consider the algorithm (R U). What if instead of saying that (U) adds one, we say that (R U) adds one? We can play the same game using this algorithm. The solved cube represents zero, (R U) represents one, (R U R U) represents two, etc. This algorithm performs a much more complicated action on the cube, so we should be able to represent more numbers. In fact, the maximum number we can represent this way is 104, and the cube re-solves itself after 105 iterations. We would say that the algorithm has *order 105*.



There are still lots of cube states left; can we do better? Unfortunately, it's only possible to get to 1259, wrapping around on the 1260th iteration. You can try this using the algorithm `R U2 D' B D'`. This has been proven to be the maximum order possible [2].

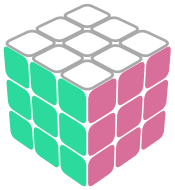
2.0.3) Branching

The next thing that a computer must be able to do is *branch*: without it we can only do addition and subtraction and nothing else. If we want to perform loops or only execute code conditionally, qter must be able to change what it does based on the state of the cube. For this, we introduce a `solved-goto` instruction.

If you perform `R U` on a cube a bunch of times without counting, it's essentially impossible for you to tell how many times you did the algorithm by *just looking* at the cube. With one exception: If you did it *zero* times, then the cube is solved and it's completely obvious that you did it zero times. Since we want qter code to be executable by humans, the `solved-goto` instruction asks you to jump to a different location of the program *only if* the cube is solved. Otherwise, you simply go to the next instruction. This is functionally equivalent to a "jump-if-zero" instruction which exists in most computer architectures.

(R U) × ???

(R U) × 0



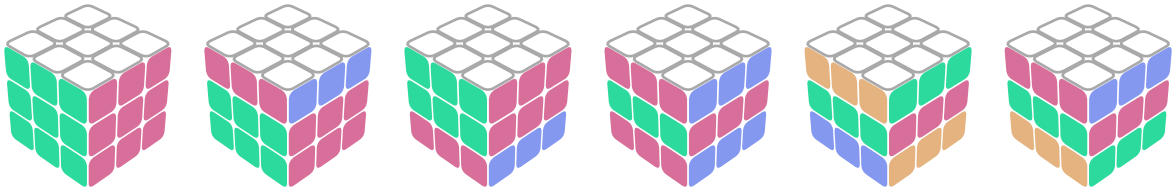
2.0.4) Multiple numbers

If you think about what programs you could actually execute with just a single number and a "jump if zero" instruction, it would be almost nothing. It would be impossible for `solved-goto` jumps to be taken without erasing all data stored on the cube. What would be wonderful is if we could represent *multiple* numbers on the cube at the same time.

Something cool about Rubik's Cubes is that it's possible for a long sequence of moves to only affect a small part of the cube. For example, we showed in the introduction an algorithm (L2 D2 L' U' L D2 L' U L') which cycles three corners. Therefore, it should be possible to represent two numbers using two algorithms that affect distinct "areas" of the cube.

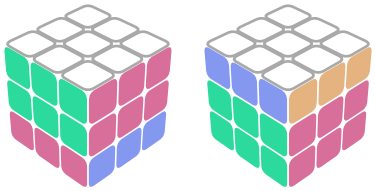
The simplest example of this are the algorithms (U) and (D'). You can see that (U) and (D') both allow representing numbers up to three, and since they affect different areas of the cube, we can represent *two different* numbers on the cube at the *same time*. We call these "registers", as an analogy to the concept in classical computing.

(0, 0) (1, 0) (0, 1) (1, 1) (3, 2) (1, 3)

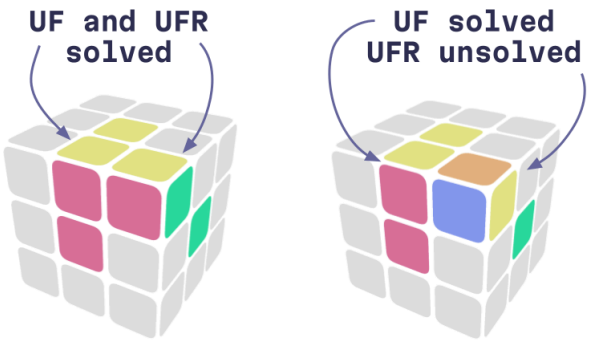


As described, `solved-goto` would only branch if the entire cube is solved, however since each algorithm affects a distinct area of the cube, it's possible for a human to determine whether a *single* register is zero, by inspecting whether a particular section of the cube is solved. Remember that "solved" means that all of the stickers are the same color as the corresponding center.

(0, ?) (?, 0)

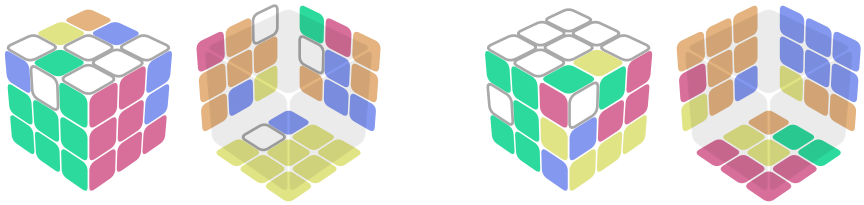


For the first cube in the above figure, it's easy to tell that the first register is zero because the entire top layer of the cube is solved. We can modify the "solved-goto" instruction to input a list of pieces, all of which must be solved for the branch to be taken, but not necessarily any more. The following illustrates a successful `solved-goto UF UFR` instruction that would require jumping to a different part of the program, as well as an unsuccessful one that would require going to the next instruction.



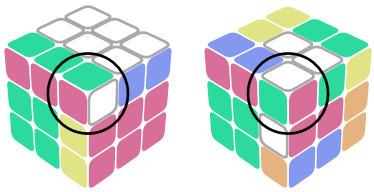
Can we do better than two registers with four states? In fact we can! If you try out the algorithms `R' F' L U' L U L F U' R` and `U F R' D' R2 F R' U' D`, you can see that they affect different pieces and both have order ninety. You may notice that they both twist the DBL corner; this is not a problem because they are independently decodable even ignoring that corner. One of the biggest challenges in the development of qter has been finding sets of algorithms with high orders that are all independently decodable. This is the fundamental problem that the Qter Architecture Solver attempts to solve, and will be discussed in later sections.

R' F' L U' L U L F U' R (1, 0) U F R' D' R2 F R' U' D (0, 1)



Another fun thing that tweaking the "solved-goto" instruction in this way allows us to do is test whether the current value of a register is divisible by a particular set of numbers. For example, returning to the register defined by RU, we can test divisibility by three by looking at the the UFR corner.

R U $(R\ U)^3$



You can see that that piece resolves itself *before* the rest of the register does, allowing us to check divisibility by three. This will be further elaborated on in Section 3.1.

All of the concepts described actually apply to other so-called "twisty puzzles", for example the Pyraminx, which is essentially a pyramid shaped Rubik's Cube. Only the notation and algorithms would have to change. For the rest of the paper, we will just look at the 3x3x3 because that is what most people are familiar with.

This is in fact all that's necessary to do things like calculating Fibonacci and performing multiplication. So now, how can we represent Qter programs?

2.1) Q language

The Q language is Qter's representation of an executable program. The file format was designed in such a way that, with only basic Rubik's Cube knowledge, a human can physically manipulate a twisty puzzle to execute a program and perform a meaningful computation.

Q files are expected to be read from top to bottom. Each line indicates an instruction, the simplest of which is just an algorithm to perform on the cube. For example:

```
Puzzles
A: 3x3

1 | U' R2
2 | L D'
...
```

The `Puzzles` declaration specifies the types of twisty puzzles used. In this example, it is declaring that you must start with a 3x3x3 cube, and that it has the name "A". The name is unimportant in this example, but becomes important when operating on multiple cubes. The instructions indicate that you must perform the algorithm `U' R2 L D'` on the Rubik's Cube. You must begin with the cube solved before following the instructions.

The Q file format also includes special instructions that involve the twisty puzzle but require additional logic. These logical instructions are designed to be simple enough for humans to understand and perform.

2.1.1) Logical instructions

- `goto <number>`

Jump to the specified line number instead of reading on to the next line. For example:

```
Puzzles
A: 3x3

1 | U' R2
2 | L D'
3 | goto 1
...
```

Indicates an infinite loop of performing (U' R2 L D') on the Rubik's Cube. After performing the algorithm, the `goto` instruction requires you to jump back to line 1 where you started.

- `solved-goto <number> <positions...>`

If the specified positions on the puzzle each contain their solved piece, then jump to the line number specified as if it was a `goto` instruction. Otherwise, do nothing and go to the next instruction. Refer to Section 2.0.4 for more details. For example:

```
Puzzles
A: 3x3

1 | U' R2
2 | solved-goto 4 UFR UF
3 | goto 1
4 | L D'
...
```

indicates performing (U' R2) and then repeatedly performing (U' R2) until the UFR corner position and UF edge position contain their solved pieces. Then, perform (L D') on the Rubik's Cube.

- `solve`

Solve the puzzle using your favorite method. Logically, this instruction zeroes out all registers on the puzzle.

- `repeat until <positions...> solved <algorithm>`

Repeat the given algorithm until the given positions contain their solved pieces. Logically, this is equivalent to

```
N   | solved-goto N+3 <positions...>
N+1 | <algorithm>
N+2 | goto N
N+3 | ...
```

but is easier to read and understand. This pattern occurs enough in Q programs that it is worth defining an instruction for it.

- `input <message> <algorithm> max-input <number>`

This instruction allows taking in arbitrary input from a user which will be stored and processed on the puzzle. To give an input, repeat the given algorithm "your input" number of times. For example:

```
Puzzles
A: 3x3

1 | input "Pick a number"
          R U R' U'
          max-input 5
...
```

To input the number two, execute the algorithm ((R U R' U') (R U R' U')) on the Rubik's Cube. Notice that if you try to execute (R U R' U') six times, the cube will return to its solved state as if you had inputted the number zero. Thus, your input number must not be greater than five, and this is shown with the `max-input 5` syntax.

If a negative input is meaningful to the program you are executing, you can input negative one by performing the inverse of the algorithm. For example, negative two would be inputted as ((U R U' R') (U R U' R')).

- `halt <message> [<algorithm> counting-until <positions...>]`

This instruction terminates the program and gives an output, and it is similar to the `input` instruction but in reverse. To decode the output of the program, repeat the given algorithm until the given positions given are solved. The number of repetitions it took to solve the pieces, along with the specified message, is considered the output of the program. For example:

```
Puzzles
A: 3x3

1 | input "Choose a number"
          R U R' U'
          max-input 5
2 | halt "You chose"
          U R U' R'
          counting-until UFR
```

In this example, after performing the input and reaching the halt instruction, you would have to repeat `U R U' R'` until the UFR corner is solved. For example, if you inputted the number two by performing

`(R U R' U') (R U R' U')`, the expected output will be two, since you have to perform `U R U' R'` twice to solve the UFR corner. Therefore, the expected output of the program is "You chose 2".

If the program does not require giving a numeric output, then the algorithm may be left out. For example:

```
Puzzles
A: 3x3

1 | halt "I halt immediately"
```

- `print <message> [<algorithm> counting-until <positions...>]`

This is an optional instruction that you may choose to ignore. The `print` instruction serves as a secondary mechanism to produce output without exiting the program. The motivation stems from the fact that, without this instruction, the only form of meaningful output is the single number produced by the `halt` instruction.

To execute this instruction, repeat the given algorithm until the positions are solved, analogous to the halt instruction. The number of repetitions this took is then the output of the print statement. Then, you must perform the inverse of the algorithm the same number of times, undoing what you just did and returning the puzzle to the state it was in before executing the print instruction. For example:

```
Puzzles
A: 3x3

1 | R U R2 B2 U L U' L' D' R' D R B2 U2
2 | print "This should output ten:"
          R U counting-until UFR UF
3 | halt "This should also output ten:"
          R U counting-until UFR UF
```

Like the `halt` instruction, including only a message is allowed. In this case, you can skip this instruction as there is nothing to do. For example:

```
Puzzles
A: 3x3

1 | print "Just a friendly debugging message :-)"
...
```

- `switch <letter>`

This instruction allows Qter to support using multiple puzzles in a single program. It tells you to put down your current puzzle and pick up a different one, labeled by letter in the `Puzzles` declaration. It is important that you do not rotate the puzzle when setting it aside or picking it back up. For example:

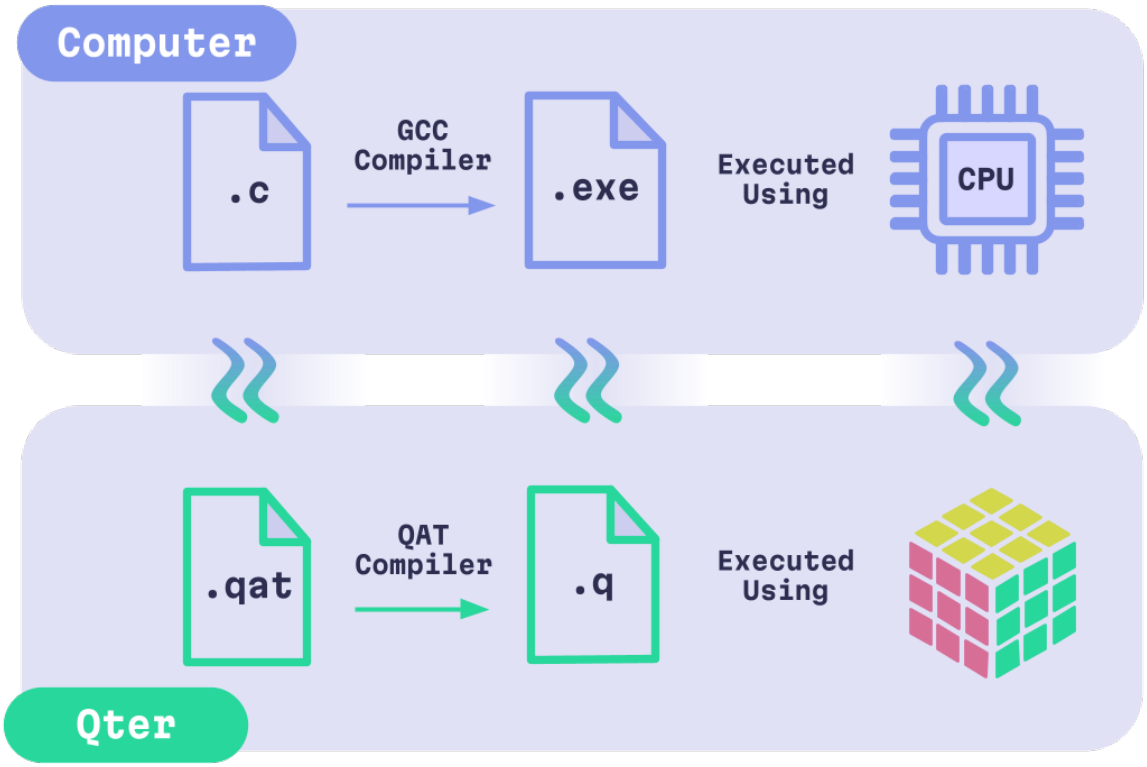
```
Puzzles
A: 3x3
B: 3x3

1 | U
2 | switch B
3 | R
...
```

This program requires two Rubik's Cubes to execute. The instructions indicate performing `U` on the first Rubik's Cube and then `R` on the second. When the program starts, you are expected to be holding

the first cube in the list. Having multiple Rubik's Cubes is helpful for when a single one doesn't provide enough storage space for what you wish to do.

2.2) QAT language

Q would be very difficult to create programs in by hand, similarly to how it is difficult to write programs in machine code directly. Therefore, we created a high-level programming language called *QAT* (Qter Assembly Text) that is designed to make it easy to write meaningful Qter programs. To run a program in a traditional programming language, you compile your source code into machine code that the computer processor then interprets and executes. The Qter compilation pipeline works similarly.



To run your first QAT program, you will first need to install Cargo (talk about installing Cargo) and then the qter compiler executable through the command line: `cargo install qter`. Once set up, create a file named `average.qat` with the following program code.

```
.registers {
    A, B <- 3x3 builtin (90, 90)
}

    -- Calculate the average of two numbers
    input "First number:" A
    input "Second number:" B
    print "Calculating average..."
sum_loop:
    add A 1
    add B 89
    solved-goto B found_sum
    goto sum_loop
found_sum:
    add A 1
divide_by_2:
```

```
    add A 89
    solved-goto A stop
    add A 89
    solved-goto A stop
    add B 1
    goto divide_by_2
stop:
    halt "The average is" B
```

To compile this program, run `qter compile average.qat` to generate `average.q`. To execute it, run `qter interpret average.q` and enter your favorite two numbers into the prompts.

2.2.1) Global variables

Every QAT program begins with a `.registers` statement, used to declare global variables named registers. The statement in the above average program declares two global registers of size 90 to be stored on a Rubik's Cube. That is, additions operate modulo 90: incrementing a register of value 89 resets it back to 0, and decrementing a register of value 0 sets it to 89.

The `builtin` keyword refers to the fact that valid register sizes are specified in a puzzle-specific preset. For the Rubik's Cube, all builtin register sizes are in src/qter_core/puzzles/3x3.txt. Unlike traditional computers, qter is only able to operate with small and irregular register sizes.

You can choose to use larger register sizes at the cost of requiring more puzzles. For example, 1260 is a valid builtin register size that needs an entire Rubik's Cube to declare. If your program wants access to more than one register, it would have to use multiple Rubik's Cubes for more memory.

```
.registers {
    A <- 3x3 builtin (1260)
    B <- 3x3 builtin (1260)
    ...
}
```

To access the remainder of a register as explained in Section 2.0.4, you can write, for example, `A%3` to access the remainder after division by three.

The `.registers` statement is also used to declare memory tapes, which help facilitate local variables, call stacks, and heap memory. This idea will be expanded upon in Section 2.3.

2.2.2) Basic instructions

The basic instructions of the QAT programming language mimic an assembly-like language. If you have already read Section 2.1, notice the similarities with QAT.

- `add <variable> <number>`

Add a constant number to a variable. This is the only way to change the value of a variable.

- `goto <label>`

Jump to a label, an identifier used to mark a specific location within code. The syntax for declaring a label follows the common convention amongst assembly languages:

```
infinite_loop:
    goto infinite_loop
```

- `solved-goto <variable> <label>`

Jump to a label if the specified variable is zero. The name of this instruction is significant in the Q file format.

- `input <message> <variable>`

Ask the user for numeric input, which will be added to the given variable.

- `print <message> [<variable>]`

Output a message, optionally followed by a variable's value.

- `halt <message> [<variable>]`

Terminate the program with a message, optionally followed by a variable's value.

2.2.3) Metaprogramming

As described, QAT is not much higher level than Q… Ideally we need some kind of construction to allow abstraction and code reuse. Due to the fact that Rubik's Cubes have extremely limited memory, we cannot maintain a call stack in the way that a classical computer would. Therefore, we cannot incorporate functions into QAT. Instead, we have a rust-inspired macro system that operates through inlining. Note that this macro system is unimplemented at the time of writing.

2.2.3.a) Defines

The simplest form of this provided by QAT is the simple `.define` statement, allowing you to define a variety of global constants.

```
.define PI 3          -- Global Integer
.define ALSO_PI $PI   -- Reference a previous define statement
.define ALSO_A A      -- Save an identifier
.define DO_ADDITION { -- Name a whole code block
    add A 10
}

add A $PI
add $ALSO_A $ALSO_PI
$DO_ADDITION
-- `A` will store the number 16
```

However, this is most likely too simple for your use case…

2.2.3.b) Macros

Macros roughly have the following syntax:

```
.macro <name> {
    (<pattern>) => <expansion>
    (<pattern>) => <expansion>
    ...
}
```

As a simple example, consider a macro to increment a register:

```
.macro inc {
    ($R:reg) => add $R 1
}
```

You would invoke it like

```
inc A
```

and it would be transformed at compile time to

```
add A 1
```

In the macro definition, `$R` represents a placeholder that any register could take the place of.

Now consider a more complicated macro, one to move the value of one register into another:

```
.macro move {
    ($R1:reg to $R2:reg) => {
        loop:
            solved-goto $R1 finished
            dec $R1
            inc $R2
            goto loop
        finished:
    }
}
```

You would invoke it like

```
move A to B
```

The word `to` is simply an identifier that must be matched for the macro invocation to compile. It allows you to make your macros read like english. This invocation would be expanded to

```
loop:
    solved-goto A finished
    dec A
    inc B
    goto loop
finished:
```

which would be expanded again to

```
loop:
    solved-goto A finished
    sub A 1
    add B 1
    goto loop
finished:
```

The expansion of `sub` will depend on the size of register A, and we'll see how to define the `sub` macro later.

Labels in macros will also be unique-ified so that if you invoke `move` twice, the labels will not conflict. This will also prevent you from jumping inside the macro invocation from outside:

```
move A to B
goto finished -- Error: the `finished` label is undefined
```

Already, we have created a powerful system for encapsulating and abstracting code, but we still have to perform control flow using manual labels and jumping. Can we extend our macro system to allow defining control flow? In fact, we can! We can define an `if` macro like

```
.macro if {
    (solved $R:reg $code:block) => {
            solved-goto $R do_if
            goto after_if
        do_if:
            $code
        after_if:
    }
}
```

and we can invoke it like

```
if solved A {
    -- Do something
}
```

which would be expanded to

```
    solved-goto A do_if
    goto after_if
do_if:
    -- Do something
after_if:
```

Here, `$code` is a placeholder for an arbitrary block of code, which allows defining custom control flow. The unique-ification of labels also covers code blocks, so the following wouldn't compile:

```
if solved A {
    goto do_if -- Error: the `do_if` label is undefined
}
```

Let's try defining a macro that executes a code block in an infinite loop:

```
.macro loop {
    ($code:block) => {
        continue:
            $code
            goto continue
        break:
    }
}
```

We can invoke it like

```
loop {
    inc A
}
```

but how can we break out of the loop? It would clearly be desirable to be able to `goto` the `continue` and `break` labels that are in the macro definition, but we can't do that. The solution is to mark the labels public, like

```
.macro loop {
    ($code:block) => {
        !continue:
            $code
            goto continue
        !break:
    }
}
```

The exclamation mark tells the compiler that the label should be accessible to code blocks inside the macro definition, so the following would be allowed:

```
loop {
    inc A

    if solved A {
        goto break
    }
}
```

However, the labels are not public to the surroundings of the macro to preserve encapsulation.

```
loop {
    -- Stuff
}
goto break -- Error: the `break` label is undefined
```

2.2.3.c) Lua Macros

For situations where macros as described before aren't expressive enough, you can embed programs written in Lua into your QAT code to enable compile-time code generation. Lets see how the `sub` macro can be defined:

```
.start-lua
    function subtract_order_relative(r1, n)
        return { { "add", r1, order_of_reg(r1) - n } }
    end
end-lua

.macro sub {
    ($R:reg $N:int) => lua subtract_order_relative($R, $N)
}
```

`lua` is a special statement that allows you to call a lua function at compile-time, and the code returned by the function will be spliced in its place. Lua macros should return a list of instructions, each of which is itself a list of the instruction name and arguments.

Here, invoking the `sub` macro will invoke the lua code to calculate what the `sub` macro should actually emit. In this example, the lua macro accesses the size of the register to calculate which addition would cause it to overflow and wrap around, having the effect of subtraction. It would be impossible to do that with simple template-replacing macros.

In general, you can write any lua code that you need to in order to make what you need to happen, happen. There are a handful of extra functions that QAT gives Lua access to.

```
big(number) -> bigint -- Takes in a standard lua number and returns a custom bigint
type that is used for register orders and instructions
order_of_reg(register) -> bigint -- Inputs an opaque reference to a register and
returns the order of that register
```

If the lua code throws an error, compilation will fail.

You can also invoke lua code in define statements:

```
.start-lua
    function bruh()
        return 5
    end
end-lua

.define FIVE lua bruh()
```

2.2.3.d) Importing

Finally, it is typically desirable to separate code between multiple files. QAT provides an `import` statement that brings all defines and macros of a different QAT file into scope, and splices any code defined in that file to the call site.

```
-- file-a.qat

.registers {
    A <- 3x3 builtin (1260)
```

```
}

add A 1
import "./file-b.qat"
thingy A

halt A
-- file-b.qat

add A 12

.macro thingy {
    ($R:reg) => {
        add $R 10
    }
}
```

Compiling and executing `file-a.qat` would print `23`.

2.2.4) Standard library

Lucky for you, you get a lot of macros built into the language! The QAT standard library is defined at src/qter_core/prelude.qat and you can reference it if you like.

`sub <register> <number>`

Subtract a number from a register

`inc <register>`

Increment a register

`dec <register>`

Decrement a register

`move <register1> to <register2>`

Zero out the first register and add its contents to the second register

`set <register1> to <register2>`

Set the contents of the first register to the contents of the second while zeroing out the contents of the second

`set <register> to <number>`

Set the contents of the first register to the number specified

`if solved <register> <{}> [else <{}>]`

Execute the code block if the register is zero, otherwise execute the `else` block if supplied

`if not-solved <register> <{}> [else <{}>]`

Execute the code block if the register is *not* zero, otherwise execute the `else` block if supplied

`if equals <register> <number> <{}> [else <{}>]`

Execute the code block if the register equals the number supplied, otherwise execute the `else` block if supplied

`if not-equals <register> <number> <{}> [else <{}>]`

Execute the code block if the register does not equal the number supplied, otherwise execute the `else` block if supplied

```
if equals <register1> <register2> using <register3> <{}> [else <{}>]
```

Execute the code block if the first two registers are equal, while passing in a third register to use as bookkeeping that will be set to zero. Otherwise executes the `else` block if supplied. All three registers must have equal order. This is validated at compile-time. The equality check is implemented by decrementing both registers until one of them is zero, so the bookkeeping register is used to save the amount of times decremented.

```
if not-equals <register1> <register2> using <register3> <{}> [else <{}>]
```

Execute the code block if the first two registers are *not* equal, while passing in a third register to use as bookkeeping that will be set to zero. Otherwise executes the `else` block if supplied. All three registers must have equal order. This is validated at compile-time. The equality check is implemented identically to `if equals ... using ...`.

```
loop <{}>
!continue
!break
```

Executes a code block in a loop forever until the `break` label or a label outside of the block is jumped to. The `break` label will exit the loop and the `continue` label will jump back to the beginning of the code block

```
while solved <register> <{}>
!continue
!break
```

Execute the block in a loop while the register is zero

```
while not-solved <register> <{}>
!continue
!break
```

Execute the block in a loop while the register is *not* zero

```
while equals <register> <number> <{}>
!continue
!break
```

Execute the block in a loop while the register is equal to the number provided

```
while not-equals <register> <number> <{}>
!continue
!break
```

Execute the block in a loop while the register is *not* equal to the number provided

```
while equals <register1> <register2> using <register3> <{}>
!continue
!break
```

Execute the block in a loop while the two registers are equal, using a third register for bookkeeping that will be zeroed out at the start of each iteration.

```
while not-equals <register1> <register2> using <register3> <{}>
!continue
!break
```

Execute the block in a loop while the two registers are *not* equal, using a third register for bookkeeping that will be zeroed out at the start of each iteration.

`repeat <number> [<ident>] <{}>`

Repeat the code block the number of times supplied, optionally providing a loop index with the name specified. The index will be emitted as a `.define` statement.

`repeat <ident> from <number1> to <number2> <{}>`

Repeat the code block for each number in the range [number1, number2)

`multiply <register1> <number> at <register2>`

Add the result of multiplying the first register with the number provided to the second register, while zeroing out the first register

`multiply <register1> <register2> using <register3>`

Multiply the first two registers, storing the result in the first register and zeroing out the second, while using the third register for bookkeeping. The third register will be zeroed out. All three registers must be the same order, which is checked at compile time.

2.3) Memory tapes

Now we're getting to the more theoretical side, as well as into a design space that we're still exploring. Things can easily change.

There are plenty of cool programs one can write using the system described above, but it's certainly not Turing complete. The fundamental reason is that we only have finite memory… For example it would be impossible to write a QAT compiler in QAT because there's simply not enough memory to even store a whole program on a Rubik's Cube. In principle, anything would be possible with infinite Rubik's Cubes, but it wouldn't be practical to give all of them names since you can't put infinite names in a program. How can we organize them instead?

The traditional solution to this problem that is used by classical computers is *pointers*. You assign every piece of memory a number and allow that number to be stored in memory itself. Each piece of memory essentially has a unique name — its number — and you can calculate which pieces of memory are needed at runtime as necessary. However, this system won't work for qter because we would like to avoid requiring the user to manually decode registers outside of halting. We allow the `print` instruction to exist because it doesn't affect what the program does and can simply be ignored at the user's discretion.

Even if we did allow pointers, it wouldn't be a foundation for the usage of infinite memory. The maximum number that a single Rubik's Cube could represent if you use the whole cube for one register is 1259. Therefore, we could only possibly assign numbers to 1260 Rubik's Cubes, which would still not be nearly enough memory to compile a QAT program.

Since our language is so minimal, we can take inspiration from perhaps the most famous barely-Turing-complete language out there (sorry in advance)… Brainfuck!! Brainfuck consists of an infinite list of numbers and a single pointer (stored externally) to the "current" number that is being operated on. A Brainfuck program consists of a list of the following operations:

- `>` Move the pointer to the right
- `<` Move the pointer to the left
- `+` Increment the number at the pointer
- `-` Decrement the number at the pointer
- `.` Output the number at the pointer

- `,` Input a number and store it where the pointer is
- `[` Jump past the matching `]` if the number at the pointer is zero
- `]` Jump to the matching `[` if the number at the pointer is non-zero

The similarity to Qter is immediately striking and it provides a blueprint for how we can support infinite cubes. We can give Qter an infinite list of cubes called a *memory tape* and instructions to move left and right, and that would make Qter Turing-complete. Now Brainfuck is intentionally designed to be a "Turing tarpit" and to make writing programs as annoying as possible, but we don't want that. For the sake of our sanity, we support having multiple memory tapes and naming them, so you don't have to think about potentially messing up other pieces of data while traversing for something else. To model a tape in a hand-computation of a qter program, one could have a bunch of Rubik's Cubes on a table laid out in a row and a physical pointer like an arrow cut out of paper to model the pointer. One could also set the currently pointed-to Rubik's Cube aside.

Lets see how we can tweak Q and QAT to interact with memory tapes. First, we need a way to declare them in both languages. In Q, you can write

```
Puzzles
tape A: 3x3
```

to mark A as a *tape* of 3x3s rather than just one 3x3. In QAT, you can write

```
.registers {
    tape X ~ A, B ← 3x3 builtin (90, 90)
}
```

to declare a memory tape X of 3x3s with the 90/90 architecture. Equivalently, you can replace the `tape` keyword with the '📼' emoji in both contexts:

```
Puzzles
📼 A: 3x3
```

```
.registers {
    📼 X ~ A, B ← 3x3 builtin (90, 90)
}
```

In Q, we need syntax to move the tape left and right, equivalent to `<` and `>` in Brainfuck. As with multiple Rubik's Cubes, tapes are switched between using the `switch` instruction, and any operations like moves or `solved-goto` will apply to the currently pointed-to Rubik's Cube.

- `move-left [<number>]`

Move the pointer to the left by the number of spaces given, or just one space if not specified

- `move-right [<number>]`

Move the pointer to the right by the number of spaces given, or just one space if not specified

In QAT, tapes can be operated on like...

```
.registers {
    📼 X ~ A, B ← 3x3 builtin (90, 90)
}

add X.A 1         -- Add one to the `A` register of the currently selected Rubik's
Cube on the `X` tape

move-right X 1    -- Move to the right
print "A is" X.A  -- Prints `A is 0` because we added one to the cube on the left
```

```
move-left X 1     -- Move to the left
print "A is" X.A  -- Prints `A is 1` because this is the puzzle that we added one to
before
```

We poo-pooed pointers previously, however this system is actually powerful enough to implement them using QAT's metaprogramming functionality, provided that we store the current head position in a register external to the tape. The following `deref` macro moves the head to a position specified in the `to` register, using the `current` register to track the current location of the head.

```
.macro deref {
    ($tape:tape $current:reg $to:reg) => {
        -- Move the head to the zero position
        while not-solved $current {
            dec $current
            move-left $tape
        }

        -- Move the head to `to`
        while not-solved $to {
            dec $to
            inc $current
            move-right $tape
        }
    }
}
```

3) Qter Architecture Solver

3.1) Introduction

Now that we understand how to write programs using Qter, how can we actually *find* sets of algorithms that function as registers? For this, it's time to get into the hardcore mathematics...

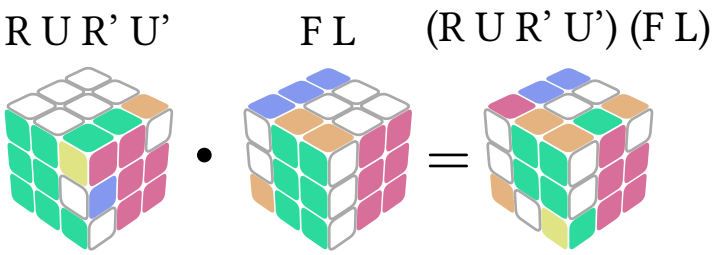
3.1.1) Group theory

First, we have to build a foundation of how we can represent Rubik's Cubes in the language of mathematics. That foundation is called *group theory*. A *group* is defined to be a *set* equipped with an *operation* (denoted ab or $a \cdot b$) that follows the following *group axioms*:

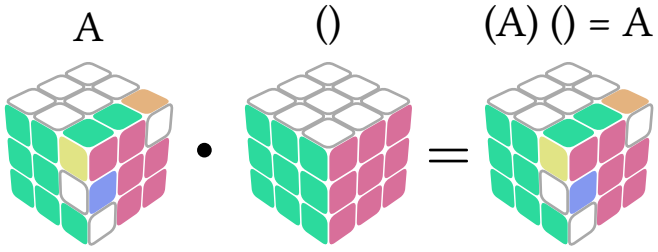
- There exists an *identity* element e such that for any element of the group a, $a \cdot e = a$.
- For all elements a, b, c, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. In other words, the operation is *associative*.
- For each a in the group, there exists a^{-1} such that $a \cdot a^{-1} = e$. In other words, every element has an *inverse* with respect to the group operation.

Importantly, commutativity is *not* required. So let's see how this definition applies to the Rubik's Cube. To form a group, we need a *set*, and for the Rubik's Cube, this set is all $4.3 \cdot 10^{19}$ possible cube states and scrambles, excluding rotations. For example, the solved state is an element of the set. If you turn the top face then that's an element of the set. If you just scramble your cube randomly and do any sequence of moves, then even that's part of the set.

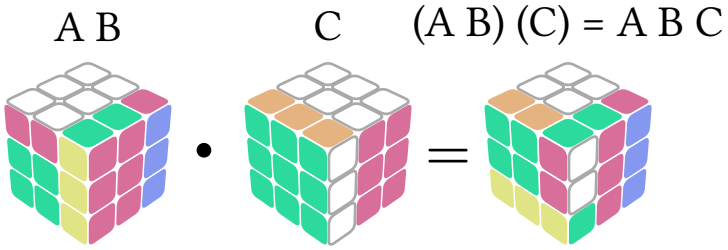
Next, we need an *operation*. For the Rubik's Cube, this will be jamming together the algorithms that reach the two cube states. We will call this operation *composition* because it is very similar to function composition.

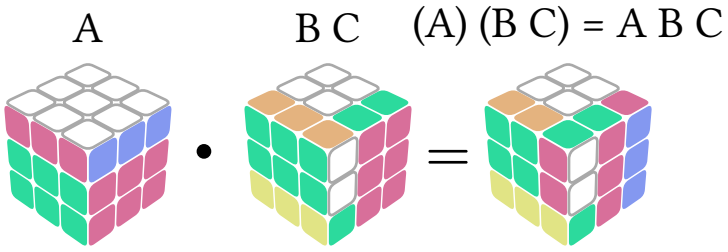


Now, let's verify that all of the group axioms hold. First, we need an identity element. This identity is simply the solved state! Lets verify this, and let A be an arbitrary scramble:



Regardless of what the first cube state is, appending the "do nothing" algorithm will lead to the same cube state. Next, lets verify associativity, letting A, B, and C be arbitrary scrambles.





Because of the nature of how jamming together algorithms works, parentheses can essentially be ignored. Therefore, the composition operation is associative. Finally we must show that every cube position has an inverse. Intuitively, we should expect an inverse to exist simply because we can undo whatever algorithm created the scramble. Here is an algorithm to find the inverse of a scramble:

```
function inverse(moves: List<Move>): List<Move> {
  reverse(moves)

  for (move in moves) {
    if move.ends_with("'") {
      remove(`'` from move)
    } else if move.ends_with("2") {
      // Leave it
    } else {
      append(`'` to move)
    }
```

```
  }

  return moves
}
```

This works because any clockwise base move X cancels with it's counterclockwise pair X' and vice versa, and any double turn X2 cancels with itself.

$$\begin{aligned}\text{R' U2 F L} \cdot \text{inverse(R' U2 F L)} &= \text{(R' U2 F L)(L' F' U2 R)} \\ &= \text{R' U2 F F' U2 R} \\ &= \text{R' U2 U2 R} \\ &= \text{R' R} \\ &= ()\end{aligned}$$

Next, it is important to distinguish a *cube state* from an *algorithm to reach that cube state*. We just described the group of Rubik's cube *algorithms* but not the group of Rubik's cube *states*. The groups are analagous but not identical: after all, there are an infinite number of move sequences that you can do, however there is only a finite number of cube states. We can say that the group of Rubik's cube *algorithms* is an *action* on the group of Rubik's cube *states*. We will explore this group of Rubik's cube states next, because it turns out that it is much more amenable to mathematical analysis and representation inside of a computer. After all, it would be problematic performance-wise if composition of Rubik's cube states was performed by concatenating potentially unbounded lists of moves, and it doesn't give us insight into the structure of the puzzle itself. To show a better way to represent a Rubik's cube state, I first have to explain…

3.1.2) Permutation groups

There are lots of other things that can form groups, but the things that we're interested in are *permutations*, which are re-arrangements of items in a set. For example, we could notate a permutation like

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 1 & 4 & 3 & 0 \end{array}$$

where the arrows define the rearrangement. Note that we can have permutations of any number of items rather than just five. We can leave out the top row of the mapping because it will always be the numbers in order, so we could notate it $2, 1, 4, 3, 0$. We can see that this permutation can also be thought of as an invertible, or *bijective*, function between the numbers $\{0, 1, 2, 3, 4\}$ and themselves.

So now, lets construct a group. The set of all permutations of a particular size, five in this example, will be the set representing our group. Then, we need an operation. Since permutations are basically functions, permutation composition can simply be function composition!

Permutation composition

$$\begin{aligned} a = &\quad 2, \quad 1, \quad 4, \quad 3, \quad 0 \\ b = &\quad 4, \quad 3, \quad 0, \quad 2, \quad 1 \\ &\quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ a \cdot b = &\ a(4), a(3), a(0), a(2), a(1) \\ = &\quad 0, \quad 3, \quad 2, \quad 4, \quad 1 \end{aligned}$$

From here, the group axioms are trivial. Our identity e is the do-nothing permutation, $0, 1, 2, 3, 4$. We know that associativity holds because permutation composition is identical to function composition

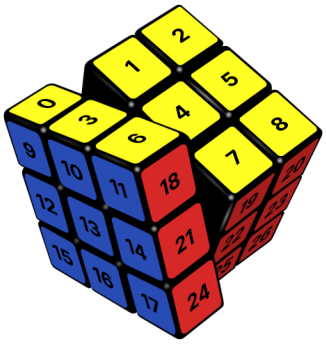
which is known to be associative. We know that there is always an inverse because permutations are *bijective* mappings and you can simply reverse the arrows to form the inverse:

$$a^{-1} = \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 2 & 1 & 4 & 3 & 0 \end{array} \rightarrow \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 4 & 1 & 0 & 3 & 2 \end{array}$$

Therefore, permutation composition satisfies all of the group axioms, so it is a group. Next, there also exists a much cleaner way to notate permutations, called *cycle notation*. The way you would write a in cycle notation is as $(0, 2, 4)(1)(3)$. Each item maps to the next item in the list, wrapping around at a closing parenthesis. The notation is saying that 0 maps to 2, 2 maps to 4, 4 maps to 0 (because of the wraparound), 1 maps to itself, and 3 also maps to itself. This is called "cycle notation" because it shows clearly the underlying cycle structure of the permutation. 0, 2, and 4 form a three-cycle and 1 and 3 both form one-cycles. It is also conventional to leave out the one-cycles and to just write down $(0, 2, 4)$.

This notation also provides a simple way to determine exactly *how many* times one can exponentiate a permutation for it to equal identity. Since a three-cycle takes three iterations for its elements to return to their initial spots, you can compose a three-cycle with itself three times to give identity. In full generality, we have to take the *least common multiple* of all of the cycle lengths to give that number of repetitions. For example, the permutation $(0, 1, 2)(3, 4, 5, 6)$ has a three-cycle and a four-cycle, and the LCM of three and four is 12, therefore exponentiating it to the twelfth power gives identity.

A permutation is something that we can easily represent in a computer, but how can we represent a Rubik's Cube in terms of permutations? It is quite simple actually...



A Rubik's Cube forms a permutation of the stickers! We don't actually have to consider the centers because they don't move, so we would have a permutation of $(9 - 1) \cdot 6 = 48$ stickers. We can define the turns on a Rubik's Cube in terms of permutations like so [3]:

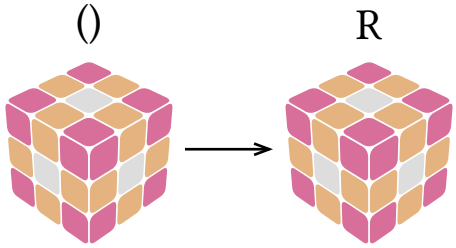
$$U = (1, 3, 8, 6)(2, 5, 7, 4)(9, 33, 25, 17)(10, 34, 26, 18)(11, 35, 27, 19)$$
$$D = (41, 43, 48, 46)(42, 45, 47, 44)(14, 22, 30, 38)(15, 23, 31, 39)(16, 24, 32, 40)$$
$$R = (25, 27, 32, 30)(26, 29, 31, 28)(3, 38, 43, 19)(5, 36, 45, 21)(8, 33, 48, 24)$$
$$L = (9, 11, 16, 14)(10, 13, 15, 12)(1, 17, 41, 40)(4, 20, 44, 37)(6, 22, 46, 35)$$
$$F = (17, 19, 24, 22)(18, 21, 23, 20)(6, 25, 43, 16)(7, 28, 42, 13)(8, 30, 41, 11)$$
$$B = (33, 35, 40, 38)(34, 37, 39, 36)(3, 9, 46, 32)(2, 12, 47, 29)(1, 14, 48, 27)$$

The exact numbers aren't actually relevant for understanding, but you can sanity-check that exponentiating all of them to the fourth gives identity, due to all of the cycles having length four. This matches our expectation of how Rubik's Cube moves should work.

Now, if we restrict our set of permutations to only contain the permutations that are reachable through combinations of $\langle U, D, R, L, F, B \rangle$ moves (after all, we can't arbitrarily re-sticker the cube), then this structure is mathematically identical — *isomorphic* — to the Rubik's Cube group. This is called a *subgroup* of the permutation group of 48 elements because the Rubik's Cube group is like its own group hidden inside that group of permutations.

It may appear as if our definition of the Rubik's cube group includes too many elements: after all, each sticker on a Rubik's cube has seven identical twins, but we're giving them different numbers and treating them as if they were unique. If there existed an algorithm that could swap two stickers of the same color, then our definition would count those as different states whereas they would really be the same state. However, we don't have to worry about this because all of the *pieces* on a cube are unique. The only way to swap two stickers would be to swap two pieces, and that would definitely produce a different cube state. Note that we don't get to make that assumption for puzzles like the 4x4x4 which have identical center pieces, however we are conveniently not writing about the 4x4x4 because our code doesn't even work for that yet 🤫 .

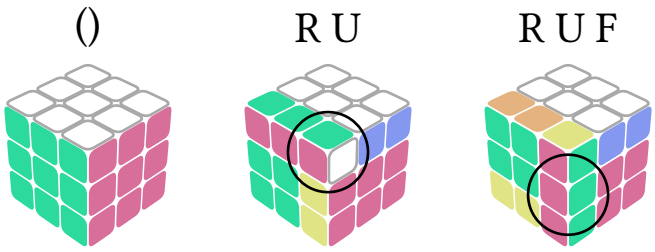
One final term to define is an *orbit*. An orbit is a collection of stickers (or whatever elements are being permuted, in full generality) such that if there exists a sequence of moves that moves one sticker in the orbit to another sticker's place, then that other sticker must be in the same orbit as the first. On a Rubik's Cube, there are two orbits: the corners and the edges. There obviously doesn't exist an algorithm that can move a corner sticker to an edge sticker's place or vice versa, therefore the corners and edges form separate orbits. Intuitively, you can find orbits of any permutation subgroup by coloring the stickers using the most colors possible such that the colors don't change when applying moves.



Excluding centers, the best we can do is two colors, and those two colors highlight the corner and edge orbits.

3.1.3) Parity and Orientation sum

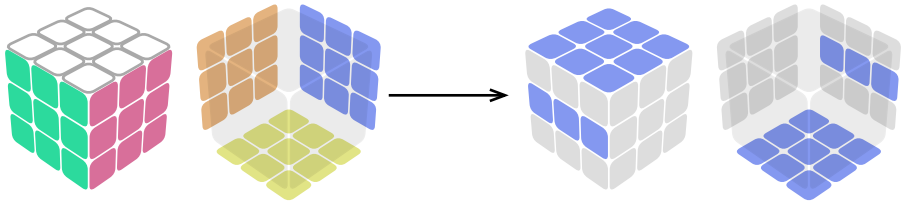
Now, we need to show some properties of how the Rubik's Cube group works. First, we would ideally like a way to take pieces into account in our representation of the Rubik's Cube group. After all, we showed in the introduction how important they are to the mechanics of the cube. What we could do is instead of having a permutation group over all of the stickers, we could have a permutation group over all of the *pieces*. There are 12 edges + 8 corners = 20 pieces on a Rubik's Cube, so we need a subgroup of the permutations on 20 elements. That's fine and dandy, but actually not sufficient to encode the full cube state. The reason is that pieces can rotate in place:



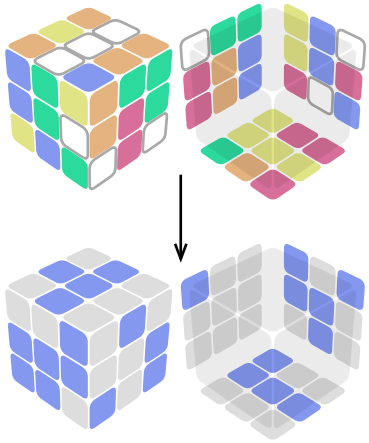
You can see that happening here, where the UFR corner is *twisted* in place in the first example and the FR edge is *flipped* in place in the second example. This shows that *just* encoding the positions of the pieces under-specifies the entire cube state, so we need to take orientation into account.

In general, any edge or corner can exist in any other edge or corner position in any orientation. So how can we encode this orientation in full generality? It's easy to tell that the UFR corner and FR edge are twisted and flipped respectively in the above examples because the pieces can be solved by simply rotating them in place. However, when the pieces are not in their solved positions, there is no way to solve them just by rotating them in place. We need some kind of reference frame to decide how to label a piece's orientation regardless of where it is on the cube. How can we define this reference frame?

Since the problem is that pieces can be unsolved, what we can do is imagine a special recoloring of the cube such that all pieces are indistinguishable but still show orientation. If the pieces aren't distinguishable, then they're *always* in their "solved positions" since you can't tell them apart. Then it's easy to define orientation in full generality. Here is a recoloring that does that:



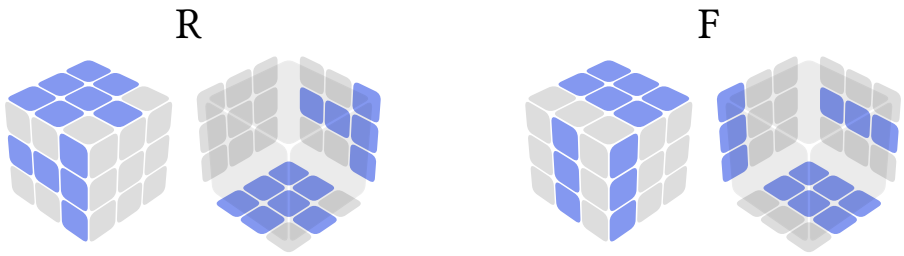
You can imagine that we are taking a Rubik's cube and replacing all of the stickers with new stickers of the respective colors. The reason that we can do this is that we already know how to represent the locations of pieces using a permutation group, so it is valid to throw out the knowledge of a piece's location while figuring out how to represent orientation. To determine the orientation of a piece on a normally colored Rubik's Cube, you can take the algorithm to get to that cube state and apply it to our specially recolored cube:



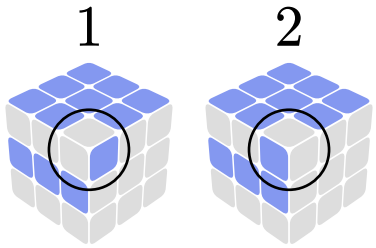
Even though the UFR corner isn't in its solved position, we can still say that the piece in the UFR position is twisted because the blue sticker isn't facing up, like it is in the recolored solved state. You would be able to "solve" that piece—make it look like the respective position in the recolored solved state—by simply rotating it in place. This gives us a reference frame to define orientation for a piece regardless of where it is located on the cube.

Note that this recoloring is entirely arbitrary and it's possible to consider *any* recoloring of the solved state such that all pieces are indistinguishable but still exhibit orientation, as long as you are consistent with your choice. However, this recoloring is standard due to its nice symmetries as well as properties we will describe in the next paragraph.

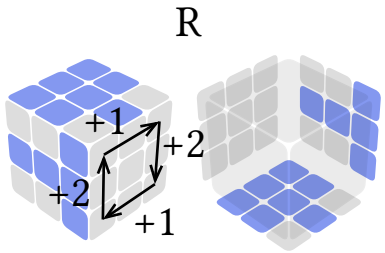
Based on this recoloring, you can see that the move set $\langle U, D, R2, F2, L2, B2 \rangle$ preserves orientation of all of the pieces, and on top of that, R and L preserve orientation of the edges but not of the corners. The moves F and B flip four edges, while R, F, L, and B twist four corners.



Note that corners actually have *two* ways of being misoriented. If the corner is twisted clockwise, we say that its orientation is one, and if it's counter-clockwise, we say that its orientation is two. Otherwise, it is zero.

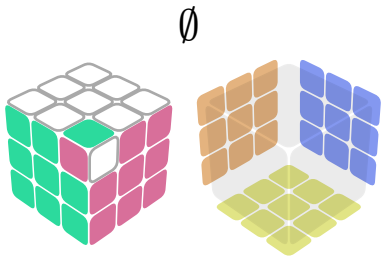


We know that F and B flip four edges, but what do R, F, L, and B do to corners? Well whatever it is, those four do the same thing because all four of those moves are symmetric to each other with respect to corners in our recoloring. Therefore, we can track what happens to the corners for just one of them.



This should make logical sense. We already know that if you apply R twice, the corners don't get twisted, and that can be seen in the figure as well. If you perform R twice, each corner will get a $+1$ twist and a $+2$ twist, which sums to three, except that three wraps around to zero.

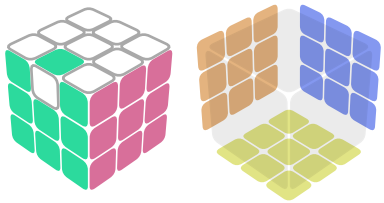
From here, we can prove that for *any* cube position, if you sum the orientations of all of the corners, you get zero. Any quarter turn about R, F, L, and B adds a total of $1 + 2 + 1 + 2 = 6$ twists to the corners, which wraps around to zero. Therefore, moves cannot change the total orientation sum so it always remains zero. This shows why a single corner twist is unsolvable on the Rubik's Cube:



The orientation sum for the corners in this position is one (one for the twisted corner plus zero for the rest), however it's impossible to apply just one twist using moves, and the corner orientation sum will always be one regardless of the moves that you do.

Similarly, we can show that the orientation sum of *edges* is also always zero. If we call the non-flipped state "zero" and the flipped state "one", then the F and B turns both flip four edges, adding $+4$ to the edge orientation sum of the cube, which wraps around to zero. Therefore, a single edge flip is unsolvable too:

$$\emptyset$$



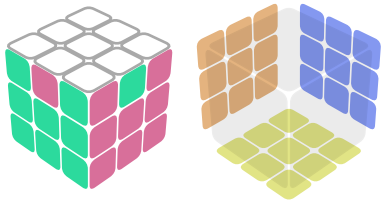
Is there anything else that's unsolvable? Actually, yes! For this to make sense, we have to think of permutations as a composition of various swaps. For example, a four-cycle can be composed out of three swaps:

$$(1,2)\cdot(1,3)\cdot(1,4) = (1,2,3)\cdot(1,4) = (1,2,3,4)$$

In general, any permutation can be expressed as a composition of swaps. So what does this have to do with Rubik's Cubes? Well a funny thing with swaps is that permutations can *only* either be expressed as a combination of an even or an odd number of swaps. This is called the *parity* of a permutation. You can see that a four-cycle has odd parity because creating it requires an odd number of swaps. Any quarter turn of a Rubik's Cube can be expressed as a four cycle of corners and a four cycle of edges, which is $3 + 3 = 6$ swaps. Overall, the permutation is even.

Therefore, a two-swap of Rubik's Cube pieces is unsolvable because creating it requires a single swap, and doing turns only does even permutations, meaning the permutation of pieces will always remain odd.

$$\emptyset$$

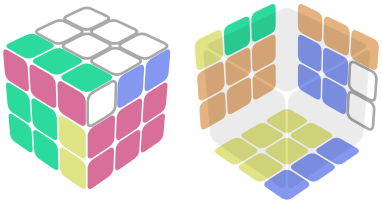


Is there any other arrangement of pieces that is unsolvable? Actually no! You can show this by counting the number of ways that you can take apart and randomly put together a Rubik's Cube, then dividing that by three because two thirds of those positions will be unsolvable due to the corner orientation sum being non-zero. Then divide by two for edge orientation sum, and then divide by two again for parity. You will see that the number you get is $4.3 \cdot 10^{19}$ which is exactly the size of the Rubik's Cube group.

3.1.4) Cycle structures

Now that we understand orientation, we can notate cube states in terms of permutation and orientation of pieces rather than just permutation of stickers. This will make the way in which the Qter Architecture Solver works easier to think about. Lets see how we can represent the RU algorithm.

R U

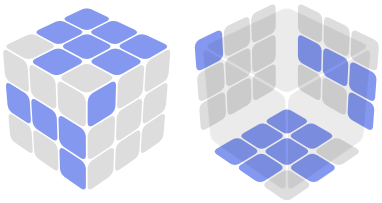


Next, lets trace where the pieces go. Instead of using numbers to represent the pieces in the cycle notation, we can simply use their names.

$$(\text{UFR})(\text{FDR}, \text{UFL}, \text{UBL}, \text{UBR}, \text{DBR})(\text{FR}, \text{UF}, \text{UL}, \text{UB}, \text{UR}, \text{BR}, \text{DR})$$

Note that I'm writing down the one-cycle of the UFR corner because we will see that it twists in place. If you would like, you can manually verify the tracing of the pieces. Next, we need to examine changes of orientation.

R U



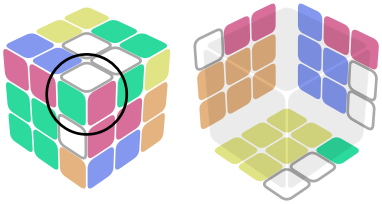
I'm going to notate orientation by writing the amount of orientation that a piece acquires above it.

$$\begin{matrix} +1 & +2 & +0 & +0 & +2 & +1 & +0 & +0 & +0 & +0 & +0 & +0 & +0 \\ (\text{UFR})(\text{FDR}, & \text{UFL}, & \text{UBL}, & \text{UBR}, & \text{DBR})(\text{FR}, & \text{UF}, & \text{UL}, & \text{UB}, & \text{UR}, & \text{BR}, & \text{DR}) \end{matrix}$$

The process of translating a cube state into cycles of pieces including orientation is known as *blind tracing* because when blind solvers memorize a puzzle, they memorize this representation. Using this representation, we can actually calculate the order of the algorithm. In the intro, we claimed that the *RU* algorithm repeats after performing it 105 times, but now we can prove it.

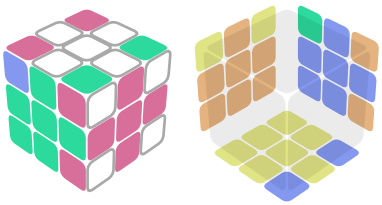
First, we have to consider how many iterations it takes for each cycle to return to solved. To find this, we have to consider both the length of the cycle and the overall orientation accrued by each piece over the length of the cycle. Lets consider the first cycle first. It has length one, meaning the piece stays in its solved location, however the piece returns with some orientation added, so it takes three iterations overall for that piece to return to solved.

$$(\text{R U})^3$$



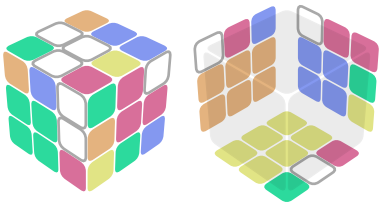
Next, let's consider the cycle of edges. They have a cycle of seven and don't accrue orientation at all, so it simply takes 7 iterations for the edges to return to solved.

$$(\mathrm{R\ U})^7$$



Finally, let's consider the cycle of corners. It has length 5, so all pieces return to their solved locations after 5 iterations, but you can see that they accrue some amount of orientation.

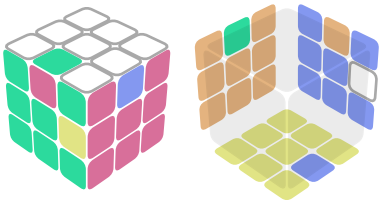
$$(\mathrm{R\ U})^5$$



How can we calculate how much orientation? Since each piece will move through each location in the cycle, it will move through each addition of orientation, meaning that all pieces will accrue the *same* orientation, and that orientation will be the sum of all orientation changes, looping around after three. The cycle has three orientation changes, $+2$, $+2$, and $+1$, and summing them gives $+5$ which loops around to $+2$. You can see in the above example that all corners in the cycle have $+2$ orientation.

It will take three traversals through the cycle for the orientation of the pieces to return to zero, so the cycle resolves itself after 15 iterations.

$$(\mathrm{R\ U})^{15}$$



Now, the *entire* cycle resolves itself once all individual cycles resolve themselves. To calculate when, we can simply take the LCM:

$$\mathrm{lcm}(3, 7, 15) = 105$$

This also clarifies what pieces we have to select as parameters for "solved-goto". We need a representative piece from every cycle that isn't redundant. We don't need to care about the 3 cycle because it is always solved whenever the 15 cycle is. We can pick any representatives from the 7 and 15 cycles, for example FDR and FR. Using those, the QAT program

```
.registers {
  A ← 3x3 (R U)
}

label:
solved-goto A label
```

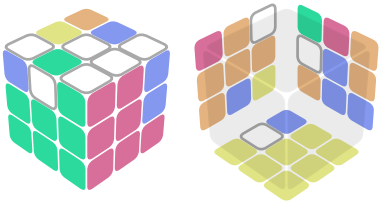
...compiles to the Q program

```
Puzzles
A: 3x3

1 | solved-goto FDR FR 1
```

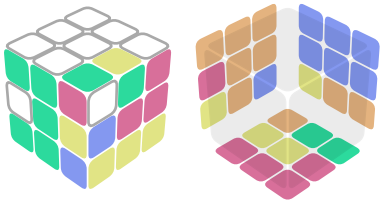
3.1.5) Orientation and parity sharing

Lets examine a real Qter architecture, for example the 90/90 one:

A = R' F' L U' L U L F U' R



B = U F R' D' R2 F R' U' D



Now let's blind-trace the cube positions:

$$A = \overset{+2}{(\text{DBL})}\overset{+1}{(\text{UF})}\overset{+1}{(\text{UFL}}\overset{+0}{,}\overset{+0}{\text{UBL}}\overset{+0}{,}\overset{+1}{\text{UBR})}\overset{+0}{(\text{UL}}\overset{+1}{,}\overset{+0}{\text{LB}}\overset{+0}{,}\overset{+0}{\text{RB}}\overset{+0}{,}\overset{+1}{\text{UB}}\overset{+0}{,}\overset{+0}{\text{LD})}$$

$$B = \overset{+1}{(\text{DBL})}\overset{+1}{(\text{UFR})}\overset{+1}{(\text{DFR}}\overset{+1}{,}\overset{+2}{\text{DFL}}\overset{+1}{,}\overset{+0}{\text{DBR})}\overset{+0}{(\text{RD})}\overset{+0}{(\text{UR}}\overset{+0}{,}\overset{+1}{\text{FL}}\overset{+0}{,}\overset{+1}{\text{DB}}\overset{+0}{,}\overset{+0}{\text{FR}}\overset{+0}{,}\overset{+0}{\text{FD})}$$

From here, we can calculate the orders of each register. A has cycles of length $3, 2, 9, 10$ with LCM 90, and B has cycles $3, 3, 9, 2, 10$ with LCM 90. However, we can see that both cycles twist the DBL corner! This is not good for the cycles being independently decodable. However, what we can do is ignore that one piece when calculating cycle lengths and performing "solved-goto" instructions. Without that shared piece, we get that A has cycles $2, 9, 10$ still with LCM 90 and B has cycles $3, 9, 2, 10$ still with LCM 90.

Why would we need to share pieces? The fundamental reason is due to the orientation and parity constraints described previously. You've seen that having a non-zero orientation sum allows the lengths of cycles to be extended beyond what they might otherwise be, however that net orientation needs to be cancelled out elsewhere to ensure that the orientation sum of the whole puzzle remains zero. For example, for the register A, the $+2$ on DBL cancels out the $+1$ on that 15 cycle.

It's possible for us to use the same piece across different registers to cancel out orientation, allowing more pieces to be used for storing data. We call this *orientation sharing*, and the pieces that are shared are called *shared pieces*. We can also use sharing to cancel out parity. For both A and B, all of the cycles that contribute to the order have even parity, meaning that parity doesn't need to be cancelled out. However if they had odd parity, then we could share two pieces that can be swapped to cancel out parity. We call that *parity sharing*.

Note that it would actually be possible for all of the DBL, UFR, UF, and RD pieces to be shared and the cycles would still work; it just happens that they aren't. If they were shared, then there could be the possibility of a shorter algorithm to produce a cycle, but at the cost of the ability to use those pieces to detect whether the register is divisible by two or three.

3.1.6) What is the Qter Architecture Solver?

You now have all of the background knowledge required to understand what the Qter Architecture Solver does. It is split into two phases:

The Cycle Combination Finder calculates what registers are possible on a Rubik's Cube by determining how cycles can be constructed and how pieces would have to be shared. One of the outputs of Cycle Combination Finder for the 90/90 architecture shown above would be something like:

```
Shared: Two corners, Two edges
A:
  - Cycle of three corners with any non-zero net orientation
  - Cycle of five edges with non-zero net orientation
B:
  - Cycle of three corners with any non-zero net orientation
  - Cycle of five edges with non-zero net orientation
```

Then the Cycle Combination Solver would take that as input and output the shortest possible algorithms that produce the given cycle structures.

Oh, and all of the theory that we just covered is generalizable to arbitrary twisty puzzles, and the Qter Architecture Solver is programmed to work for all of them. However, we will stick to the familiar Rubik's Cube for our explanation.

3.2) Cycle Combination Finder

You saw an early example of utilizing cycles as registers within the cube: the U algorithm can be defined addition by 1. This example is a good introduction, but it only allows for a single cycle of four states.

Ideally we would have more states and multiple cycles. The Cycle Combination Finder (CCF) finds all 'non-redundant' cycle combinations, those which cannot be contained within any larger combinations. A 90/80 (90 cycle and 80 cycle) is redundant, since 90/90 is also possible. It contains all of the 90/80 positions, as well as additional positions that are not possible with 90/80, such as (81,81).

To define some terms, we will let the set of cycles that represent a register be the *cycle combination* of that register. For example, the cycle combination of RU is the set of the 3, 7, and 15 cycles that make it up. An *architecture* is the set of cycle combinations of all registers, as well as the set of shared pieces that make the registers possible to realize on the cube given the orientation and parity constraints. For the purpose of the CCF, we don't need to know exactly *which* pieces need to make up each cycle or are shared. We only need the number of pieces for each orbit that are shared, and the number and orientation sum of pieces in each cycle. Figuring out which pieces are the best to use is the job of the Cycle Combination Solver.

3.2.1) Beginning with primes

To begin constructing architectures for a puzzle, we must begin by finding which individual cycles are possible to create. We begin by looking at primes. For large primes and their powers, generally 5 or up, we will be able to create a cycle that is the length of that prime power only if there is an orbit of pieces greater than or equal to that prime power. The 3x3 has an orbit of 12 edges, so the prime powers 5, 7, and 11 will fit, but 13, 25, and 1331 are too large.

For smaller primes, generally just 2 and 3, we may be able to make a more compact cycle using orientation. Instead of cycling 3 corner pieces, we can just twist a single corner, since corners have an orientation of period 3. A power of a small prime p^k will fit if there exists a number $m \leq k$ and an orbit with at least p^m pieces, and the power deficit can be made up by orienting, meaning that p^{k-m} divides the orientation period of the orbit. For example, 16 will fit since there are at least 8 edges, and we can double the length of the 8-cycle using a 2-period orientation.

Following this logic, the prime powers that fit on a 3x3 are: 1, 2, 3, 4, 5, 7, 8, 9, 11, 16.

3.2.2) Generalizing to composites

We then combine the prime powers to find all integer cycle combinations that will fit on the puzzle. Each prime power is assigned a minimum piece count, which is the minimum number of pieces required to construct that cycle. For large primes, such as 5, this is just the value itself. For the smaller primes it is p^m as shown above, replaced by 0 if $p^m = 1$. This replacement is done since a cycle made purely of orientation could be combined with one made of purely permutation. If there is a 5-cycle using 5 edges, we can insert a 2-cycle for 'free' by adding a 2-period orientation.

Given these minimum piece counts, we recursively multiply all available powers for each prime (including p^0), and exit the current branch if the piece total exceeds the number of pieces of the puzzle.

For example, an 880 cycle will not fit on the 3x3. The prime power factorization is 16, 5, and 11 which have minimum piece counts of 8, 5, and 11 respectively, adding to 24. The 3x3 only has 20 pieces so this is impossible. However, a 90 cycle may fit. The prime powers of 90 are 2, 9, and 5, which have minimum piece counts 0, 3, and 5. These add to 8, much lower than the 20 total pieces. It is important to note that this test doesn't guarantee that the cycle combination will fit, just that it cannot yet be ruled out.

3.2.3) Combining multiple cycles

Once all possible cycle orders are found, we search for all non-redundant architectures. We will generate the cycle combinations in descending order, since any architecture is equivalent to a descending one. For example, 10/20/40 is the same as 40/20/10.

First, we have to generate the list of potentially possible sets of orders of registers in an architecture, which we do by simply trying every possible set of cycle combinations that we discovered in the previous step, and pruning all values with minimum piece sums greater than the number of pieces on the puzzle, and that don't have registers in descending order. This does not guarantee that the architecture in the list can be created, but it is true that every architure that can be created is in the list.

To test if a set of orders fits on the puzzle, we decompose each order into its prime powers, and try placing each power into each orbit. For the 3x3 there are 2 orbits: corner pieces and edge pieces. For example, to test if 90/90 fits, we decompose it into prime power cycles of 2, 9, 5, 2, 9 and 5. Note that for the purpose of fitting all of the cycles onto the puzzle, we don't need to remember which cycle belongs to which register. We recursively place each cycle into each orbit, failing if there is not enough room in any orbit for the current power. This begins by trying to place the first 2-cycle in the corner orbit, and passing to the 9-cycle, then once that recursion has finished, trying to place the 2-cycle in the edge orbit and passing forward.

If all cycles get placed into an orbit, then we have found a layout that fits, and any pieces left-over can be considered shared. However, we still need to perform a final check to ensure that parity and orientation are accounted for by the shared pieces. If this check passes, we log the architecture. Otherwise it fails and we continue the search.

After a successful architecture has been found, it can be used to exit early for redundant combinations: If all possible architectures from the current branch of the search would be redundant to a successful combination, we exit and continue at the next step of the previous level. Once all possible outputs have been found, we can remove all redundant cycle combinations that we weren't able to remove during search and return from the Cycle Combination Finder.

3.3) Cycle Combination Solver

The Cycle Combination Finder of the Qter Architecture Solver finds the non-redundant cycle structures of each register in a Qter architecture. We are not done yet—for every cycle structure, we need to find an algorithm that, when applied to the solved state, yields a state with that cycle structure.

That is, we need to solve for the register's "add 1" operation. Once we have that, all other "add N"s can be derived by repeating the "add 1" operation N times and then shortening the algorithm using an external Rubik's Cube solver.

The Cycle Combination Solver adds two additional requirements to this task. First, it solves for the *shortest*, or the *optimal* algorithm that generates this cycle structure. This is technically not necessary, but considering that "add 1" is observationally the most executed instruction, it greatly reduces the overall number of moves needed to execute a *Q* program. Second, of all solutions of optimal length, it chooses the algorithm easiest to physically perform by hand, which we will discuss in a later section that follows.

In order to understand how to optimally solve for a cycle structure, we briefly turn our attention to an adjacent problem: optimally solving the Rubik's Cube.

We thank Scherpius [4] for his overview of the ideas in these next few sections.

3.3.1) Optimal solving background

First, what do we mean by "optimal" or "shortest"? We need to choose a *metric* for counting the number of moves in an algorithm, and there are a variety of ways to do so. In this paper, we will use what is known as the *half turn metric*, which means that we consider U2 to be a single move. An alternative choice would be the *quarter turn metric* which would consider U2 to be two moves, however that is less common in the literature and we won't use it in this paper.

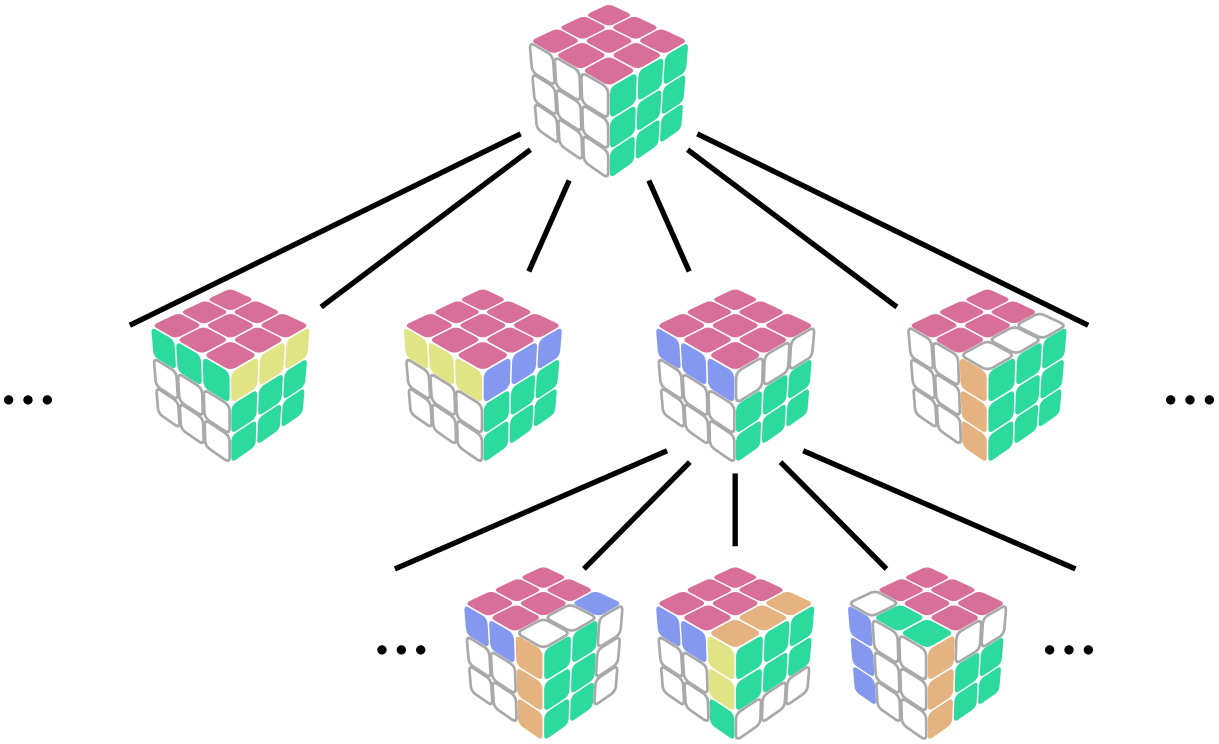
In an optimal Rubik's Cube solver, we are given a random position, and we must find the shortest algorithm that brings the Rubik's Cube to the solved state. Analogously, the Cycle Combination Solver starts from the solved state and finds the shortest algorithm that brings the puzzle to a position with our specified cycle structure. The only thing that's fundamentally changed is something trivial — the goal condition. We bring up optimal *solving* because this allows us to reuse its techniques which have been studied for the past 30 years [5].

It would be reasonable to expect there to be a known structural property of the Rubik's Cube that makes optimal solving easy. Indeed, to find a *good* solution to the Rubik's Cube, the technique of Kociemba's algorithm [6] cleverly utilizes a specific subgroup to solve up to 3900 individual position per second *near* optimally [7]. However, we want to do better than that.

Unfortunately, to find an *optimal* solution, the only known approach is to brute force all combinations of move sequences until the Rubik's Cube is solved. To add some insult to injury, Demaine [8] proved that optimal $N \times N \times N$ cube solving is NP-complete. However, this doesn't mean we can't optimize the brute force approach. We will discuss a variety of improvements that can be made, some specific to the Cycle Combination Solver only, but unless there is a significant advancement in group theory relating to the problem it is solving, the runtime is necessarily going to be exponential.

3.3.2) Tree searching

A more formal way to think about the Cycle Combination Solver is to think of the state space as a tree of Rubik's Cube positions joined by the 18 moves. The number of moves that have been applied to any given position is simply that position's corresponding level in the tree. We will refer to these positions as *nodes*.



Our goal is now to find a node with the specified cycle structure at the *topmost* level of the tree, a solution of the optimal move length. Those familiar with data structures and algorithms will think of the most obvious approach to this form of tree searching: breadth-first search (BFS). BFS is an algorithm that explores all nodes in a level before moving on to the next one. Indeed, BFS guarantees optimality, and works in theory, but not in practice: extra memory is needed to keep track of child nodes that are yet to be explored. At every level, the number of nodes scales by a factor 18, and so does the extra memory needed. At a depth level i.e. sequence length of just 8, BFS would require storing 18^9 depth-9 nodes or roughly 200 billion Rubik's Cube states in memory. This is clearly not practical; we need to do better.

We now consider a sibling algorithm to BFS: depth-first search (DFS). DFS is an algorithm that explores all nodes as deep as possible before backtracking. It strikes our interest because the memory overhead is minimal; all you need to keep track of is the path taken to reach a node, something that can be easily managed during the search. However, because we explore nodes depth-first, it offers no guarantee about optimality, so we still have a problem.

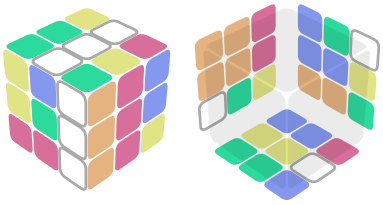
A simple modification to DFS can make it always find the optimal solution. We tweak the DFS implementation so that it explores up until a specified depth, testing whether each node at this depth is a solution, without exploring further. We repeatedly run this implementation at increasing depth limits until a solution *is* found. Put simply, you do a DFS of depth 1, then of depth 2, and so on. This idea is known as iterative-deepening depth-first search (IDDFS), a hybrid of a breadth-first and depth-first search. IDDFS does repeat some work each iteration, but the cost is always small relative to the last depth because the Rubik's Cube search tree grows exponentially. The insignificance of the repeat work is further exacerbated given that even more time is spent at the last depth running the test for a solution. Because the majority of the time is spent at the last depth d, the asymptotic time complexity of $O\left(18^d\right)$ in Big O notation is actually identical to BFS while solving the memory problem. We will gradually improve this time complexity bound throughout the rest of this section.

3.3.3) Pruning

IDDFS solves the memory issue, but is lacking in speed because tree searching is still slow. The overwhelming majority of paths explored lead to no solution. What would be nice is if we could somehow know whether all paths that continue from a given node are dead ends without having to check by brute-force.

For this, we introduce the idea of a *pruning table*. For any given Rubik's Cube position, a pruning table tells you a lower bound on the number of moves needed to reach a Cycle Combination Solver solution. Suppose we are running IDDFS until depth 12, we've done 5 moves so far, and we have reached this nodfe.

R' U2 L' D' R'



If we *query* the pruning table and it says that this position needs at least 8 moves to reach a Cycle Combination Solver solution, we know that this branch is a dead end. 5 moves done so far plus 8 left is 13, which is more than the 12 at which we plan to terminate. Hence, we can avoid having to search this position any longer.

The use of pruning tables in this fashion was originated by Korf [5] in his optimal Rubik's Cube solver. He observed the important requirement that pruning tables must provide *admissible heuristics* to guarantee optimality. That is, they must never overestimate the distance to a solution. If in the above example, the lower bound was wrong and there really was a solution in 12 moves, then the heuristic would prevent us from finding it. Combining IDDFS and an admissible heuristic is known as Iterative Deepening A* (IDA*).

How are we supposed to store all 43 quintillion positions of the Rubik's Cube in memory? Well, we don't: different types of pruning tables solve this problem by sacrificing either information or accuracy to take up less space. Hence, pruning tables give an admissible heuristic instead of the exact number of moves needed to reach a Cycle Combination Solver solution.

Loosely speaking, pruning tables can be thought of as a form of meet-in-the-middle search, more generally known as a space—time trade-off [9]. Even when running the Cycle Combination Solver on the same puzzle, we *must* generate a new pruning table for every unique cycle structure. It turns out this is still worth it. In general, we can characterize the effectiveness of a pruning table by its expected admissible heuristic, p. Pruning tables reduce the search depth of the tree because they have the effect of preventing searching the last p depths, and the improvements are dramatic because the number of nodes at increasing depths grows exponentially. But there is no free lunch: we have to pay for this speedup by memory.

We are left with a need to examine the asymptotic time complexity IDA*. An IDA* search to depth limit d is similar to an IDDFS search to depth limit $d - p$, implying a time complexity of IDA* is $O\left(18^{d-p}\right)$ (recall how the last depth is the dominating factor). One might even be eagar to consider these two searches exactly equivalent, but Korf describes a perhaps surprising discrepancy: the number of nodes visited by IDA* is empirically far greater, up to a magnitude of two. Nodes with large pruning values are quickly pruned, while nodes with small pruning values survive to spawn more nodes. Thus, IDA* search is biased in favor of smaller heuristic values, and the expected admissible heuristic is actually lesser.

Next we will conjecture that p is logarithmically correlated to the number of states the pruning table can store, which we denote as the amount of memory used m. We first assume the branching factor b is constant and then notice the minimum depth that is stored in the pruning table is approximately $\lfloor \log_b m \rfloor$. Since there are exponentially more states at the last depth, p is negligibly less than this depth; hence, $p \sim \log_b m$. In reality, the branching factor is not constant and always less than its theoretical value, eventually converging to zero. The pruning table distribution in general is nontrivial to analyze. This implies our estimate of $p \sim \log_b m$ is an egregious overestimate of the actual average pruning value, but we proceed with this approximation because IDA* is biased in favor of smaller heuristic values. These two biases cancel each other out to some extent.

As such, $O\left(18^{d-p}\right) = O\left(18^{d-\log_{18} m}\right) = O\left(\frac{18^d}{m}\right)$. Empirically and analytically, doubling the size of the pruning table halves the CPU time required to perform a search.

3.3.4) Pruning table design

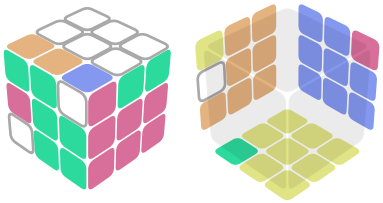
The larger the admissible heuristic, the better the pruning, and the lesser the search depth. So, we need to carefully design our pruning tables to maximize:

- how much information we can store within a given memory constraint; and
- the value of the admissible heuristic

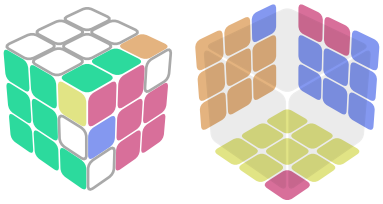
3.3.4.a) Symmetry reduction

Symmetry reduction is the most famous way to compress pruning table entries. We thank Kociemba [10] for his excellent explanations of symmetry reduction on his website. Take a good look at these two cube positions below:

A = F U F' U'



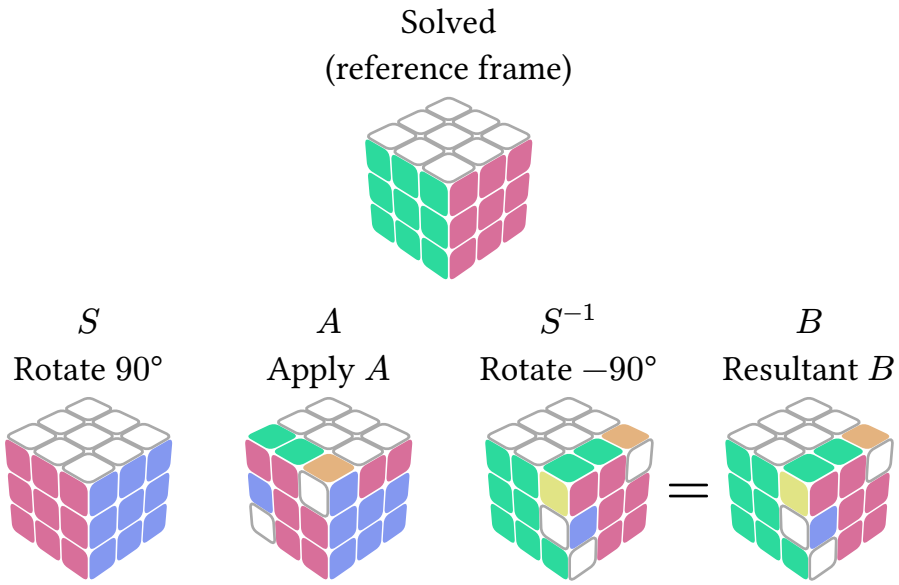
B = R U R' U'



They are different but they are *basically* identical. If you replace red with blue, blue with orange, orange with green, green with red, you will have transformed A into B. Because these two cube positions have the exact same structure of pieces, they need the same number of moves to reach a Cycle Combination Solver solution.

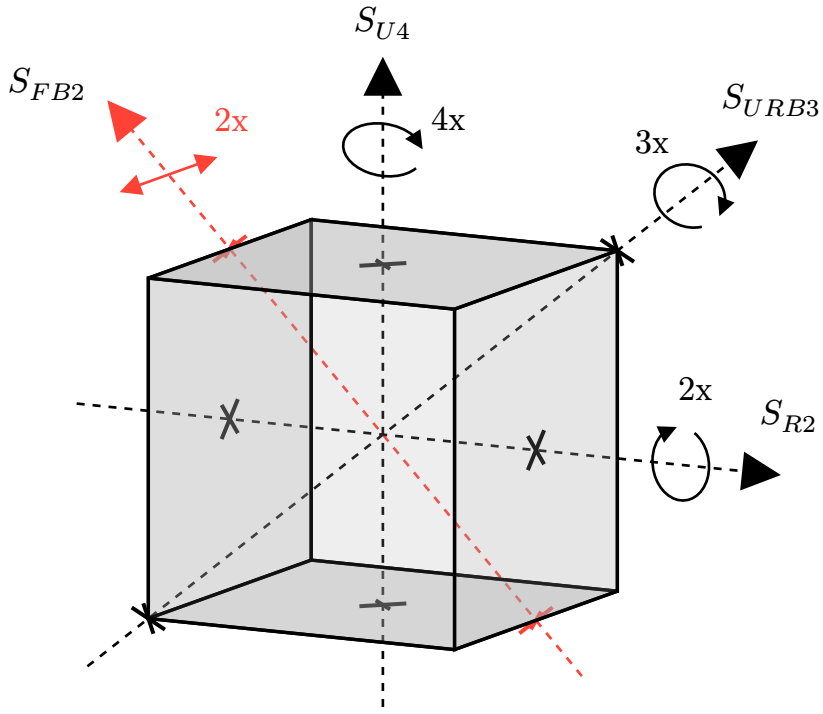
We call such positions *symmetrically equivalent*. If we really wanted to be serious about pruning table compression, what we can do is store a single representative of all symmetrically equivalent cubes because they would all share the same admissible heuristic value, and keeping a separate entry for each of these positions is a waste of memory.

Defining symmetrically equivalent cubes by figuring out an arbitrary way to recolor the cube is a very handwavy way to think about it, nor is it very efficient. The more mathematically precise way to define symmetrically equivalent cubes is with permutations. Two cube positions A and B are symmetrically equivalent if there exists a symmetry S of the cube such that $SAS^{-1} = B$, where the S operations are spatial manipulations the whole cube. We can prove that A and B are symmetrically equivalent using this model:



In group theory, SAS^{-1} is called a *conjugation* of A by S—we first perform the symmetry, apply our desired permutation, and then perform the inverse of the symmetry to restore the original reference frame. The symmetries of arbitrary polyhedra themselves form a group, called a *symmetry group*, so we can guarantee an S^{-1} element exists.

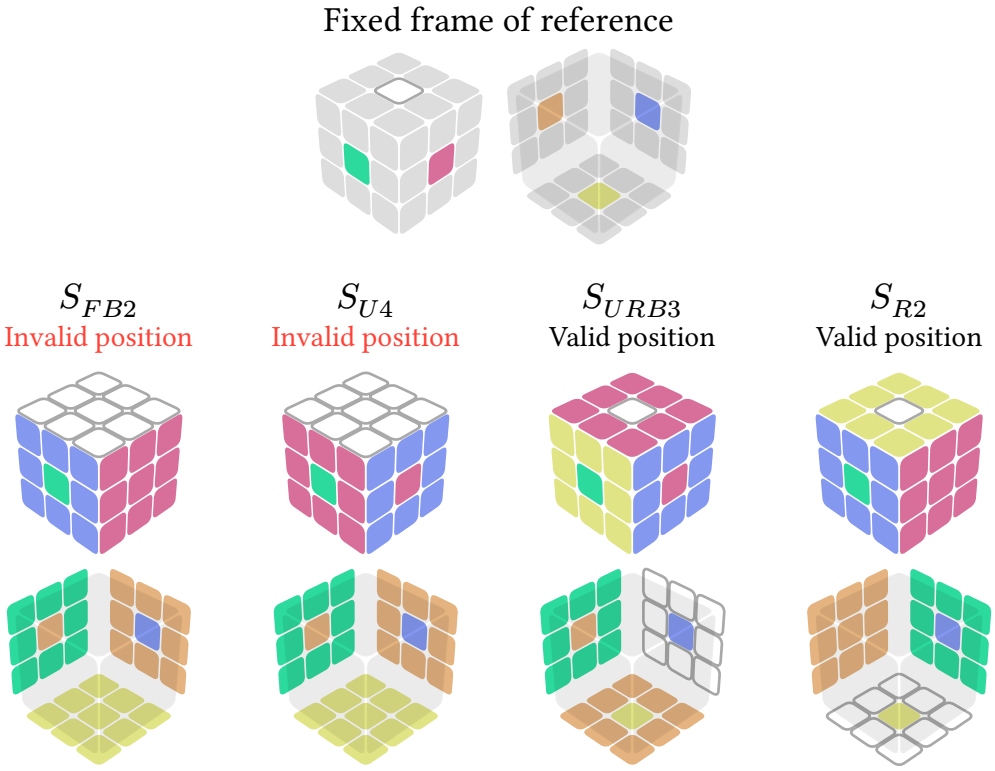
Symmetry reduction compresses the pruning table by the number distinct symmetries—all possible values of S—of the cube, so how many are there? The symmetry group of the cube M consists of 24 rotational symmetries and 24 *mirror* symmetries, for a total of 48 distinct symmetries. You can think of the mirror symmetries by imagining holding a Rubik's Cube position in a mirror to get a mirror image of that position. In this reflectional domain, we again apply the 24 rotational symmetries. We illustrate one (of very many) ways to uniquely construct all of these symmetries, with the mirror symmetry highlighted in red.



The 48 symmetries of the cube

$$M = \left\{ (S_{URB3})^a \cdot (S_{R2})^b \cdot (S_{U4})^c \cdot (S_{FB2})^d \mid a \in \{0,1,2\}, b \in \{0,1\}, c \in \{0,1,2,3\}, d \in \{0,1\} \right\}$$

We discussed how symmetry conjugation temporarily changes a position's frame of reference before subsequently restoring it. Without any further context this would be fine, but in programming we efficiently represent a Rubik's Cube position by treating the centers as a fixed reference frame to avoid storing their states. This optimization is critical for speed because it makes position composition faster and minimizes data overhead. The ensuing caveat is that we *must* always refer to a fixed frame of reference, so we have to rethink how symmetry conjugation works. The solution is simple, and the established theory still holds: we define the change of reference frame as a *position* such that, when composed with the solved state, it transforms the pieces around the fixed frame of reference.



The takeaway is in the observation that every symmetry position has the centers in the same spatial orientation.

Notice that the S_{FB2} and S_{U4} symmetries are invalid positions with this fixed reference frame—the latter because of the parity constraint, and the former because the mirror image produces a reflectional coloring. *This does not matter* because the inconsistencies are un-done when S^{-1} is applied; thus the conjugation SAS^{-1} always results in a valid position.

48 symmetries is already quite a lot, but we can still do better. If we can show that both an arbitrary Rubik's Cube position and its inverse position require the same number of moves to reach a Cycle Combination Solver solution, we can once again store a single representative of the two positions and further compress the table by another factor of 2. We call this *antisymmetry*.

Let us prove that our presumption is true.

1. Let P and S be defined as sequences such that $P\ S$ is an optimal solution to the Cycle Combination Solver.
2. We take the inverse of $P\ S$ to get $S^{-1}P^{-1}$ of the same sequence length, which is still an optimal solution to the Cycle Combination Solver. Taking the inverse of the "add 1" operation (which is P

S) is the "sub 1" operation; changing your frame of reference to think of "sub 1" as "add 1" yields another way to construct the exact same register.

3. We conjugate $S^{-1}P^{-1}$ with S to get $S(S^{-1}P^{-1})S^{-1} = P^{-1}S^{-1}$ of the same sequence length. It turns out that conjugate elements in a permutation group exhibit the same cycle structure, hence this is also an optimal solution to the Cycle Combination Solver. To understand why, we simplify the problem and examine the general case of two conjugate elements in a permutation group A and ABA^{-1}. If permutation B takes element x to y, then ABA^{-1} takes element $A(x)$ to $A(y)$. Indeed,

$$(ABA^{-1})(A(x)) = A(B(A^{-1}(A(x)))) = A(B(x)) = A(y)$$

So every cycle $(x_1, x_2, ..., x_n)$ of B is taken to the cycle $(A(x_1), A(x_2), ..., A(x_n))$ of ABA^{-1}. Viewing permutations as bijective maps of its elements, conjugation only relabels the elements moved by B. It does not change the cycle lengths nor how many cycles there are. We apply this corollary with $A = S$ and $B = S^{-1}P^{-1}$.

4. We have shown that if P S is an optimal solution to the Cycle Combination Solver then so is $P^{-1}S^{-1}$. S and S^{-1} are the same sequence length; thus, the positions reached by any arbitrary P and by P^{-1} starting from the solved state require the same number of moves to reach an optimal Cycle Combination Solver solution. This completes our proof.

Symmetry and antisymmetry reduction comes with a cost. During IDA* search, every position must be transformed to its "symmetry and antisymmetry" representative before using it to query the pruning table. To do so we conjugate the position by the 48 symmetries and the inverse by the 48 antisymmetries to explore all the possible representatives. To identify the representative position after each conjugation, we look at its raw binary state representation and choose the lexicographic minimum (i.e. the minimum comparing byte-by-byte). Multiple symmetries may produce the representative position, however that is okay because at no point do we actually care about which symmetry conjugation did so; the result is still the same.

The symmetry and antisymmetry reduction algorithm as described so far would be slow—we need to perform 96 symmetry conjugations, and each is about as expensive as two moves. We use the following trick described by Rokicki [11]: instead of computing the full conjugation for every symmetry conjugation, we compute the elements one-at-a-time. We take the least possible value for the first element of all the symmetry conjugations and filter for the ones that give us that value. Then, we compute all the second symmetry conjugation elements, find the least possible value for that, and so on. This optimization usually only ends up performing a single full symmetry conjugation.

3.3.4.b) Pruning table types

The Cycle Combination Solver uses a separate pruning table per the puzzle orbits. For the Rubik's Cube, that means one pruning table for the corners and one for the edges. To get an admissible heuristic for an individual position, we query each pruning table based on the states of the position's corresponding orbits and take the maximum value. A brief example: if querying a Rubik's Cube state returns 3 on the corners pruning table and 5 on the edges pruning table, then its admissible heuristic is the maximum of the two, 5. We established that larger heuristic values are better, and the optimality guarantee still stands because each individual pruning table is already admissible.

Generating a pruning table for an orbit is done in two phases. First, we enumerate every single position of the orbit and mark solutions of the Cycle Combination Solver. Then, we search the Rubik's Cube tree but from these solution states instead of from the solved state, and storing the amount of moves required to reach each state found as the admissible heuristic.

The Cycle Combination Solver supports four different types of pruning tables: the exact pruning table, the approximate pruning table, the cycle structure pruning table, and the fixed pruning table. They are dynamically chosen at runtime based on a maximum memory limit option.

We defer our discussion of pruning table types for a later revision.

Finally, the Cycle Combination Solver generates the pruning tables and performs IDA* search at the same time. It would not be very efficient for the Cycle Combination Solver to spend all of its time generating the pruning tables only for the actual searching part to be easy, so it balances out querying and generation; starting from an uninitialized pruning table, if the number of queries exceeds the number of set values by a factor of 3, it pauses the search to generate a deeper layer of that pruning table and then continues.

3.3.4.c) Pruning table compression

The Cycle Combination Solver supports three different data compression types: no compression, nxopt compression, and tabled asymmetric numeral systems (tANS) compression. They are dynamically chosen at runtime based on a maximum memory limit option.

We defer our discussion of pruning table compression for a later revision.

3.3.5) IDA* optimizations

We employ a number of tricks to improve the running time of the Cycle Combination Solver's IDA* tree search.

3.3.5.a) SIMD

We enhance the speed of puzzle operations through the use of puzzle-specific SIMD on AVX2 and Neon instruction set architectures. Namely, the `VPSHUFB` instruction on AVX2 and the `tbl.8`/`tbl.16` instructions on Neon perform permutation composition in one clock cycle, enabling for specialized SIMD algorithms to compose two Rubik's Cube states [12] and test for a Cycle Combination Solver solution [13]. They have both been disassembled and highly optimized at the instruction level. Additionally, the puzzle-specific SIMD uses compacted representations optimized for the permutation composition instructions. For example, it uses a representation of a Rubik's Cube state that can fit in a single `YMM` CPU register on AVX2 and in the `D` and `Q` CPU registers on Neon.

Pruning table generation also uses puzzle-specific SIMD. To generate a pruning table on the corners orbit, we need to use a different Rubik's Cube representation because we don't want to waste CPU caring about what happens to edges. So, every orbit has its own specialized SIMD representation and SIMD algorithm modifications.

We leave the precise details at the prescribed references; we defer our discussion of how the SIMD algorithms work for a later revision.

3.3.5.b) Canonical sequences

At every increasing depth level of the IDA* search tree we explore 18 times as many nodes. We formally call this number the *branching factor*—the average number of child nodes visited by a parent node. A few clever observations can reduce the branching factor.

We observe that we never want to rotate the same face twice. For example, if we perform R followed by R', we've just reversed the move done at the previous level of the tree. Similarly if we perform R followed by another R, we could have simply done $R2$ straight away. In general, any move should not be followed by another move in the same *move class*, the set of all move powers. This reduces the branching factor of the child nodes from 18 for all 18 moves to 15. Additionally, we don't want to search both RL and LR because they commute, and result in the same net action. So, we assume that R

(or $R2, R'$) never follows L (or $L2, L'$), and in general, we only permit searching distinct commutative move classes strictly in a single order only. Move sequences that satisfy these two conditions are called *canonical sequences*. Canonical sequences are special because these two conditions make it easy to check if a move sequence in the search tree is redundant.

What does the second condition reduce our branching factor from 15 to? We start by counting the number of canonical sequences at length N, denoted a_n, using a recurrence relation. We consider the last move of the sequence M_1, the second to last move M_2, and the third to last move M_3. The recurrence relation can be constructed by analyzing two cases:

- Case 1: M_1 and M_2 do not commute.

  In this case, a_n is simply a_{n-1} multiplied by the number of possibilities of M_1. Since M_1 and M_2 do not commute, M_1 cannot be M_2 (-1) nor its opposite face (-1). Therefore, M_1 must be one of $6 - 1 - 1 = 4$ move classes, or one of the $4 * 3 = 12$ possible moves. We can establish that the first component in the recurrence relation for a_n is $12a_{n-1}$.

- Case 2: M_1 and M_2 commute.

  We need to be careful to only count M_1 and M_2, one time so we count them in pairs. In this case, a_n is simply a_{n-2} multiplied by the number of strictly ordered (M_1, M_2) pairs. There are 3 pairs of commutative move classes: $FB, UD,$ and RL. We have to discard one of these pairs because M_3 necessarily commutes with the move classes in one of these pairs since the union of all of these pairs is every move. Such a canonical sequence where the subsequence $M_3M_2M_1$ all commute cannot exist because one of those moves will always violate the strict move class ordering. For example, if M_1 is L and M_2 is R, then there is no possible option for M_3 that makes the full sequence a canonical sequence.

  Each move class in each pair can perform three moves, which implies that each pair contributes $3 * 3 = 9$ possible moves. Overall we find this number to be $(3 - 1) * 9 = 18$ possible moves. We can establish that the second component in the recurrence relation for a_n is $18a_{n-2}$.

a_n can be thought of as the superposition of these two cases with the base cases $a_1 = 18$ and $a_2 = 243$ (exercise to the reader: figure out where these come from). Hence, $a_n = 12a_{n-1} + 18a_{n-2}$ for $n > 2$. The standard recurrence relation can be solved as follows:

$$
\begin{aligned}
&r^n = 12r^{n-1} + 18r^{n-2} \\
&r^{n-2}(-r^2 + 12r + 18) = 0 \\
&r = \frac{-12 \pm \sqrt{12^2 - 4(-1)(18)}}{2(-1)} \\
&r_{1,2} = 6 \pm 3\sqrt{6} \\
&a_n = Ar_1^{n-2} + Br_2^{n-2} = \frac{A}{r_1^2}r_1^n + \frac{B}{r_2^2}r_2^n \\
&\begin{cases} a_1 = 18 \\ a_2 = A \quad + B & = 243 \\ a_3 = Ar_1 + Br_2 = 12a_2 + 18a_1 = 3240 \end{cases} \\
&\text{Solve for } A \text{ and } B \\
&\cdots \\
&a_n \simeq 1.362(13.348)^n + 0.138(-1.348)^n
\end{aligned}
$$

The $1.362(13.348)^n$ term dominates $0.138(-1.348)^n$ as n approaches infinity; our new branching factor is approximately 13.348!

It turns out that a_n is not an exact bound on the number of distinct positions at sequence length N but merely an upper bound. This is because the formula overcounts, and the actual number is always lower: it considers canonical sequences that produce equivalent states such as $R2\ L2\ U2\ D2$ and $U2\ D2\ R2\ L2$ as two distinct positions. It turns out it is extremely nontrivial to describe and account for these equivalences, to the point where it's not worth doing so: at shallow and medium depths, a_n roughly stays within 10% of the actual distinct position count. The Cycle Combination Solver considers the extra work negligible and searches equivalent canonical sequences anyways. The Big O time complexity of IDA* can be realized as $O\left(\frac{13.348^d}{m}\right)$, an improvement over $O\left(\frac{18^d}{m}\right)$ from Section 3.3.2.

The Cycle Combination Solver uses an optimized finite state machine to perform the canonical sequence optimization.

3.3.5.c) Sequence symmetry

We use a special form of symmetry reduction during the search we call *sequence symmetry*, first observed by Rokicki [14] and improved by our implementation. Some solution to the Cycle Combination Solver $ABCD$ conjugated by A^{-1} yields $A^{-1}(ABCD)A = BCDA$, which we observe to be a rotation of the original sequence as well as a solution to the Cycle Combination Solver by the properties of conjugation discussed earlier. Repeatedly applying this conjugation:

$$\begin{aligned} & A^{-1}(ABCD)A = BCDA \\ \Rightarrow\ & B^{-1}(BCDA)B = CDAB \\ \Rightarrow\ & C^{-1}(CDAB)C = DABC \\ \Rightarrow\ & D^{-1}(DABC)D = ABCD \end{aligned}$$

forms an equivalence class based on all the rotations of sequences that are all solutions to the Cycle Combination Solver. The key is to search a single representative sequence in this equivalence class to avoid duplicate work.

Similarly to symmetry conjugation, we choose the representative as the lexicographically minimal sequence on a move-by-move basis (with a move class ordering relation defined). Unlike symmetry conjugation, we don't manually apply all sequence rotations to find the representative; rather, we embed sequence symmetry as a modification to the recursive IDA* algorithm such that it only ever searches the representative sequence. We do this by observing that if a *representative sequence* starts with move A, then every other move cannot be lexicographically lesser than it. If this observation were to be false, we could keep on rotating the sequence until the offending move is at the beginning of the sequence, and since that move is lexicographically lesser than A that sequence rotation would be the true representative. This contradicts the initial *representative sequence* assumption. We permit moves that are lexicographically equal to A (i.e. in the same move class) but change the next recursive step to repeat the logic on the move *after* A. The overall effect is that the IDA* algorithm only visits move sequences such that no later subsequence is lexicographically lesser than the beginning of the move sequence. This suffices for the complete sequence symmetry optimization.

The modification described is not yet foolproof. The sequence $ABABCAB$ would technically be valid as there is no later subsequence lesser than the beginning, but the actual lexicographically minimal representative is the $ABABABC$ sequence rotation. The "later subsequence" of the true representative wraps around from the end to the beginning. So, extra care must be taken at the last depth to manually account for the wrapping behavior. We only apply this to the last depth, so sequences like $ABABCABC$ are still searched by the next depth limit of IDA*.

We can extend our prior definition of canonical sequences to include sequence symmetry as a third condition. How does sequence symmetry affect the number of canonical sequences at depth N? Because a sequence of length N has N sequence rotations, sequence symmetry logically divides the total number of nodes visited by N, but only in the best case. The canonical sequence $R\ U\ R\ U\ R\ U$ only has 2 members in its sequence rotational equivalence class, not 6, so the average value to divide by is actually a bit less than N. It follows that the average number of canonical sequences at depth N (and the IDA* asymptotic time complexity) is bound by $\Omega\left(\frac{13.348^n}{mn}\right)$ and $O\left(\frac{13.348^n}{m}\right)$. Testing has shown this number to typically be right in the middle of these two bounds.

Furthermore, we take advantage of the fact that the optimal solution sequence *almost never* starts and ends with commutative moves. We claim that the IDA* algorithm *almost never* needs to test $AB\ ...\ C$ such that A and C commute for a solution. The proof is as follows.

We first observe that if $AB\ ...\ C$ is a solution, then $CAB\ ...$ is also a solution by a sequence rotation. This tells us that A and C cannot be in the same move class or else they could be combined to produce the shorter solution $DB\ ...$. Such a shorter solution would have been found at the previous depth limit, implying that $AB\ ...\ C$ never would have been explored, making this situation an impossibility. This also tells us that A also cannot be in a greater move class than C because $CAB\ ...$ would be a lexicographically lesser than $AB\ ...\ C$, contradicting our earlier proof that IDA* only searches the lexicographically minimal sequence rotation (the representative). Therefore, A must be in a lesser move class than C.

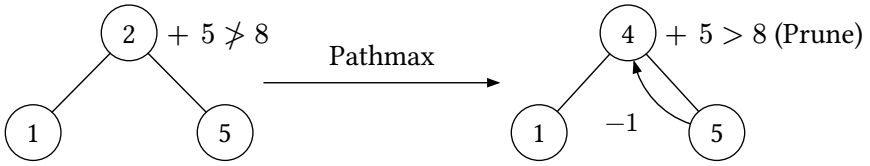
If $CAB\ ...$ is a solution, then $ACB\ ...$ is also a solution because A and C commute. By the transitive property, if $AB\ ...\ C$ is a solution, then so is $ACB\ ...$. Both of these sequences are independently searched and tested as a solution because there is no direct "commutative move ordering" or sequence symmetry relation between them. This is redundant work; we choose to discard the $AB\ ...\ C$ case. This completes our proof.

This optimization only applies to the last depth in IDA*, so it only prevents running the test to check if a node is a solution and does not affect the time complexity. It turns out to be surprisingly effective at reducing the average time per node because most of the time is spent at the last depth.

We alluded to an edge case when we said "*almost never*." If B doesn't exist, or if every move from B ... commutes with A and C, then this optimization will skip canonical sequences where every move commutes with each other; for example $F\ B$ on the Rubik's Cube. The number of skipped sequences is so small that we have the bandwidth to manually search and test these sequences for solutions before running IDA*.

3.3.5.d) Pathmax

We use a simple optimization described by Mérõ [15] called *pathmax* to prune nodes with large child pruning heuristics. When a child node has a large pruning heuristic, we can set the current node cost to that value minus one and re-prune to avoid expanding the remaining child nodes. This larger heuristic is still admissible because it is one less than a known lower bound, and the current node is one move away from all of its child nodes. This is only effective when the heuristics are *inconsistent*, or, in this case, when the pruning table entries are the minimum of two or more other values. With exact pruning tables only, this optimization will never run because the entries are perfect heuristics that cannot exhibit this type of discrepency.



IDA* pathmax at depth $= 5$, depth limit $= 8$

3.3.5.e) Parallel IDA*

Our last trick is to enhance IDA* through the use of parallel multithreaded IDA* (PMIDA* [16]). PMIDA* runs in two phases. In the first phase, we use BFS to explore the state space to a shallow depth, maintaining a queue of all of states at the last search depth. In the second phase, we use a thread pool to run IDA* in parallel for every state in that queue, utilizing of all of the CPU cores on the host machine. To uphold the optimality guarantee, PMIDA* synchronizes the threads using a barrier that triggers when they have all completed exploring the current level. It can be thought of as a simple extension to the familiar IDA* algorithm.

There have been many parallel IDA* algorithms discussed in literature; how do we know PMIDA* is the best one? We take advantage of the special fact that the Cycle Combination Solver starts searching from the solved state. In order to understand this, we compare the total Rubik's Cube position counts with the Rubik's Cube position counts that are unique by symmetry.

Rubik's Cube position counts [17]

Depth	Count	Branching factor
0	1	NA
1	18	18
2	243	13.5
3	3240	13.333
4	43239	13.345
5	574908	13.296
6	7618438	13.252
7	100803036	13.231
8	1332343288	13.217
9	17596479795	13.207

Rubik's Cube position counts unique by symmetry + antisymmetry [17]

Depth	Count	Branching factor
0	1	NA
1	2	2
2	8	4
3	48	6
4	509	10.604
5	6198	12.177
6	80178	12.936
7	1053077	13.134
8	13890036	13.190
9	183339529	13.199

Recall that our theoretical branching factor is 13.348. In the table of Rubik's Cube position counts, the branching factor roughly matches this number. However, at the shallow depths of the table of Rubik's Cube position counts unique by symmetry + antisymmetry, our branching factor is much less because there are duplicate positions when performing moves from the solved state. Intuitively, this should make sense: the Rubik's Cube is not scrambled enough to start producing unique positions. It is easy to pick out two sequences of length two that are not unique by symmetry; for example $R2\,U$ and $R2\,F$. The branching factor converges to its theoretical value as the Rubik's Cube becomes more scrambled because symmetric positions become more rare. In fact, it was shown by Qu [18] that scrambling the Rubik's Cube can literally be modelled as a Markov chain (it's almost indistinguishable from a random walk of a graph). Hence, it is unlikely for two random move sequences of the same length to produce positions equivalent by symmetry. We know that such collisions *do* happen because the branching factor doesn't actually reach the 13.348 value, but we consider them negligible.

The effectiveness of the PMIDA* algorithm stems from combining all of these observations. When our initial shallow BFS search is done, we filter out the many symmetrically equivalent positions from the queue to avoid redundant work before we start parallelizing IDA*. The savings are incredibly dramatic: at depth 4, for example, we symmetry reduce the number of nodes from 43239 to 509. This is a reduction by ~ 84.9, a factor that is close to the familiar 96 (the number of symmetries + antisymmetries). Once we do that, and the cube starts to become sufficiently scrambled, we are confident to claim that each IDA* thread worker explores their own independent regions of the search space and duplicates a negligible amount of work.

We make note that there are almost always going to be more positions in the queue to parallelize than available OS threads. We use an optimized thread pool work stealing algorithm for our multithreaded implementation.

We squeeze out our last bit of juice by overlapping pruning table memory latency with the computation. It has been empirically observed that random access into the pruning table memory is the dominating factor for Rubik's Cube solvers. Modern processors include prefetching instructions that tell the memory system to speculatively load a particular memory location into cache without stalling the execution pipeline to do so. Our PMIDA* implementation uses a technique described by Rokicki [19] called *microthreading* to spend CPU time on different subsearches while waiting for the memory to come to a query. It splits up each thread into eight "slivers" of control. Each sliver calculates a pruning table query memory address, does a prefetch, and moves on to the next sliver. When that sliver gets control again, only then does it reference the actual memory. By handling many subsearches simultaneously, microthreading minimizes the CPU idle time.

How does PMIDA* affect the asymptotic time complexity? We established in Section 3.3.5.c an upper bound of $O\left(\frac{13.348^d}{m}\right)$. The time required by PMIDA* can be computed by adding the time of the first and second phases. In the first phase the time required for the BFS is $O\left(13.348^{d_1}\right)$ where d_1 is the aforementioned shallow depth. In the second phase we symmetry reduce at the shallow depth, split the work across t independent threads, and ignore nodes before depth d_1. The time required is $O\left(\left(\frac{13.348^d}{ms} - 13.348^{d_1}\right)/t\right)$ where s is the number of symmetries + antisymmetries. The PMIDA* time complexity is thus $O\left(13.348^{d_1} + \left(\frac{13.348^d}{ms} - 13.348^{d_1}\right)/t\right)$, but we consider d_1 to be very small and s to be a negligible constant. As such the final time complexity becomes $O\left(\frac{13.348^d}{mt}\right)$. We can apply the exact same logic to our lower bound, and we get $\Omega\left(\frac{13.348^d}{dmt}\right)$.

3.3.6) Larger twisty puzzles

The overwhelming majority of our research has been within the realm of the Rubik's Cube, and so far, we have yet to run the Cycle Combination Solver on non-Rubik's Cube twisty puzzles. While we are confident all of our theory generalizes to larger twisty puzzles (with minor implementation detail differences [11]), there is a practical concern we expect to run into.

Optimally solving the 4x4x4 Rubik's Cube has been theorized to take roughly as much time as computing the minimum number of moves to solve any 3x3x3 Rubik's Cube [20], which took around 35 CPU-years [7]. It may very well be the case that the Cycle Combination Solver, even with all its optimization tricks, will never be able to find a solution to a Cycle Combination Finder cycle structure for larger twisty puzzles. Thus, we are forced to sacrifice optimality in one of three ways:

- We can write *multiphase* solvers for these larger puzzles. Multiphase solvers are specialized to the specific puzzle, and they work by incrementally bringing the twisty puzzle to a "closer to solved" state in a reasonable number of moves. However, designing a multiphase solver is profoundly more involved compared to designing an optimal solver. This approach is unsustainable because it is impractical and difficult to write a multiphase solver for every possible twisty puzzle.

- We can resort to methods to solve arbitrary permutation groups. We speculate that the most promising method of which may be to utilize something called a strong generating set [21]. The GAP computer algebra system implements this method in the `PreImagesRepresentative` function as illustrated in . The algorithms produced by the strong generating sets can quickly become large. In the future, we plan to investigate the work of Egner [22] and apply his techniques to keep the algorithm lengths in check.
- When all other options have been exhausted, we must resort to designing our cycle structure algorithms by hand. This approach would likely follow the blindfolded twisty puzzle solving method of permuting a three or five pieces at a time. Contrary to popular belief, the blindfolded solving method is simple, and it is generalizable to arbitrary twisty puzzles.

3.3.7) Movecount Coefficient Calculator

The Cycle Combination Solver's solutions are only optimal by *length*, but not by *easiness to perform*. Meaning, if you pick up a Rubik's cube right now, you would find it much harder to perform $B2\ L'\ D2$ compared to $R\ U\ R'$ despite being the same length because this algorithm requires you to awkwardly re-grip your fingers to make the turns. This might seem like an unimportant metric, but remember: we want Qter to be human-friendly, and the "add 1" instruction is observationally the most executed one.

Thus, the Cycle Combination Solver first finds *all* optimal solutions of the same length, and then utilizes our rewrite of Trang's Movecount Coefficient Calculator [23] to output the solution easiest to physically perform. The Movecount Coefficient Calculator simulates a human hand turning the Rubik's Cube to score algorithms by this metric. For non-Rubik's cube Cycle Combination Solver puzzles, we favor algorithms that turn faces on the right, top, and front of the puzzle, near where your fingers would typically be located.

3.3.8) Re-running with fixed pieces

The Cycle Combination Solver as described so far only finds the optimal solution for single register for a Qter architecture given by the Cycle Combination Finder. Now we need to re-run the Cycle Combination Solver for the remaining registers to find their optimal solutions.

Re-running the Cycle Combination Solver brings about one additional requirement: the pieces affected by previously found register algorithms must be fixed in place. We do this to ensure incrementing register A doesn't affect the state of register B; logically this kind of side-effect is nonsensical and important to prevent. The moves performed while incrementing register A *can* actually move these fixed pieces around whereever they want—what only matters is that these pieces are returned to their original positions. In other words, all of the register incrementation algorithms in a Qter architecture must commute.

Fixing pieces also means we can no longer use symmetry reduction because two symmetrically equivalent puzzles may fix different sets of pieces.

How can we be so sure that the second register found is the optimal solution possible? Yes, while the Cycle Combination Solver finds the optimal solution given the fixed pieces constraint, what if a slightly longer first register algorithm results in a significantly shorter second register algorithm? In this sense it is extremely difficult to find the provably optimal Qter architecture because of all of these possiblities. The Cycle Combination Solver does not concern itself with this problem, and it instead uses a greedy algorithm. It sorts the Cycle Combination Finder registers by their sizes (i.e. the number of states) in descending order. We observe that the average length of the optimal solution increases as more pieces on the puzzle are fixed because there are more restrictions. Solving each cycle structure in this order ensures that registers with larger sizes are prioritized with shorter algorithms because they are more likely to be incremented in a Q program than smaller sized registers.

4) Conclusion

In this article, we gave a comprehensive description of Qter from the perspective of a user, as well as from the perspective of the underlying mathematics and algorithms. If you read the whole thing, you now have the necessary background knowledge to even contribute to Qter. You've probably figured out that Qter is useful as nothing more than a piece of art or as an educational tool, but it's fulfilled that role better than we could have ever imagined.

Our journey with Qter is not even close to over, but given our track record at recruiting people to help us, yours probably is. We hope that we were able to give you the "WOW!" factor that we felt (and are still feeling) while putting this thing together. We're just a bunch of randos, and we built Qter out of knowledge scoured from Wikipedia, scraps of advice from strangers, and flashes of creativity and inspiration. We hope that we have inspired *you* to find your own Qter to obsess over for years.

A GAP programming

We provide an example run of GAP solving the random scramble $F\ L'\ D'\ B2\ U'\ B'\ U\ B2\ R2\ F'\ R2\ U2\ F'\ R2\ F\ U2\ B'\ R2\ F'\ R\ B2$ in just over two seconds using the strong generating set method.

```
gap> U := ( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19);;
gap> L := ( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35);;
gap> F := (17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11);;
gap> R := (25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24);;
gap> B := (33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27);;
gap> D := (41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40);;
gap> random_scramble := F*L^-1*D^-1*B^2*U^-1*B^-1*U*B^2*R^2*F^-1*R^2*U^2*F^-1*R^2*F
*U^2*B^-1*R^2*F^-1*R*B^2;;
gap> cube := Group(U, L, F, R, B, D);;
gap> generator_names := ["U", "L", "F", "R", "B", "D"];;
gap> hom := EpimorphismFromFreeGroup(cube:names:=generator_names);;
gap> ext_rep := ExtRepOfObj(PreImagesRepresentative(hom, random_scramble));;
gap> time;
2180
gap> for i in Reversed([1..Length(ext_rep) / 2]) do
>     Print(generator_names[ext_rep[i * 2 - 1]]);
>     count := ext_rep[i * 2];
>     if count in [-2, 2] then
>         Print("2");
>     elif count in [-3, 1] then
>         Print("'");
>     else
>         Print(" ");
>     fi;
>     Print(" ");
> od;
U  B2 R2 F  B' R' B  R  F  R  F' R' U' D  R  D' F' U  L  F2 U  L' U2 F  D  F' D' L  U
L2 U' B  L  B' U' L' U' L' B' U' B  U' L  U  L' U  L  U  F  U' F' L  U  F  U' F' L' U
F' U' L' U  L  F  L  U2 L' U' L  U' L' U2 F  U  R  U' R' F' U' F  R  U  R' F' L' B'
U2 B  U  L
```

Bibliography

[1] D. Wang, "Rubik's Cube Move Notation." [Online]. Available: https://jperm.net/3x3/moves

[2] M. Hedberg, *On Rubik's Cube*. KTH Royal Institute of Technology, 2010, pp. 65–79.

[3] “Analyzing Rubik's Cube with GAP.” [Online]. Available: https://www.math.rwth-aachen.de/homes/GAP/WWW2/Doc/Examples/rubik.html

[4] J. Scherphuis, “Computer Puzzling.” [Online]. Available: https://www.jaapsch.net/puzzles/compcube.htm

[5] R. Korf, “Finding optimal solutions to Rubik's cube using pattern databases,” 1997, *AAAI Press*.

[6] H. Kociemba, “Two-Phase Algorithm Details.” [Online]. Available: https://kociemba.org/math/imptwophase.htm

[7] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, “The diameter of the rubik's cube group is twenty,” 2014, *siam REVIEW*.

[8] E. Demaine, S. Eisenstat, and M. Rudoy, “Solving the Rubik's Cube Optimally is NP-complete,” 2018, *Schloss Dagstuhl – Leibniz-Zentrum für Informatik*. doi: 10.4230/LIPICS.STACS.2018.24.

[9] “Space–time tradeoff.” [Online]. Available: https://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff

[10] H. Kociemba, “Equivalent Cubes and Symmetry.” [Online]. Available: https://kociemba.org/cube.htm

[11] T. Rokicki, “architecture.md.” [Online]. Available: https://github.com/cubing/twsearch/blob/0dced6e55f5612609a54c75056d00535fadee0c8/docs/architecture.md

[12] A. Chaudhary, [Online]. Available: https://github.com/ArhanChaudhary/qter/blob/8d2cbcb5338250cd25c132678b838d0316f502f9/src/phase2/src/puzzle/cube3/avx2.rs#L207

[13] A. Chaudhary, [Online]. Available: https://github.com/ArhanChaudhary/qter/blob/8d2cbcb5338250cd25c132678b838d0316f502f9/src/phase2/src/puzzle/cube3/avx2.rs#L304

[14] T. Rokicki, *Support reduction by rotation of sequences in ordertree*. [Online]. Available: https://github.com/cubing/twsearch/commit/7b1d62bd9d9d232fb4729c7227d5255deed9673c

[15] L. Mérõ, “A Heuristic Search Algorithm with Modifiable Estimate,” 1984.

[16] B. Mahafzah, “Parallel multithreaded IDA* heuristic search: algorithm design and performance evaluation,” 2011, *Taylor & Francis*.

[17] T. Scheunemann, “God's Algorithm out to 15f*.” [Online]. Available: http://forum.cubeman.org/?q=node/view/201

[18] Y. Qu, T. Rokicki, and H. Yang, “Rubik's Cube Scrambling Requires at Least 26 Random Moves,” 2024. [Online]. Available: https://arxiv.org/abs/2410.20630

[19] T. Rokicki, [Online]. Available: https://github.com/cubing/twsearch/blob/0dced6e55f5612609a54c75056d00535fadee0c8/src/cpp/solve.cpp#L111

[20] T. Rokicki, “God's number is....” [Online]. Available: https://www.speedsolving.com/threads/gods-number-is.30231/post-997686

[21] “Strong generating set.” [Online]. Available: https://en.wikipedia.org/wiki/Strong_generating_set

[22] S. Egner and M. Püschel, “Solving puzzles related to permutation groups,” *Association for Computing Machinery*. [Online]. Available: https://doi.org/10.1145/281508.281611

[23] V. Trang, “Movecount Coefficient Calculator: Online Tool To Evaluate The Speed Of 3x3 Algorithms.” [Online]. Available: https://www.speedsolving.com/threads/movecount-coefficient-calculator-online-tool-to-evaluate-the-speed-of-3x3-algorithms.79025/