Would you say you're more likely to buy a print copy of this book if it had the Purdue bell tower on it?[1]

[1]Julian Herzog (Website), CC BY 4.0 <https://creativecommons.org/licenses/by/4.0>, via Wikimedia Commons

Purdue Hackers[2]

presents

a journal of things done over summer 2025 (and sometimes the fall, but we don't talk about that timeline overrun)

SIGHORSE 2025

many authors

cover art by Cynthia, Amber, and Kart
illustrations by Amber and authors

purdue university
west lafayette, IN

first presented on December 6, 2025
at SPILL[3]

thank you, sigbovik, for existing so I could copy the title page and copyright page style off you.

[2] https://www.purduehackers.com/

[3] https://spill.purduehackers.com/

Table of Contents

Foreword

Hi! I'm Kart. I came up with SIGHORSE, made the website, and (sometimes) reviewed submissions from authors. I was, of course, helped immeasurably by many people without whom SIGHORSE would not be possible, but ideally you've already seen their names before this article.

I'd like to talk about SIGHORSE for a bit.

There are three parts to this foreword:

1. Why the word "horse"? Why is it relevant? What does it mean to Purdue Hackers?
2. Why work on this journal? What forces made us want to create SIGHORSE?
3. Can we see the process of making the cover? Surely it wasn't too much work.

1. Bringing forth a longing for horsing into the world

> Purdue Hackers: a community of students who collaborate, learn, and build kick-ass technical projects.
>
> — Purdue Hackers website, but I removed the emojis

Since forever, horses have been intertwined with the Purdue Hackers brand.

Discord[1] archaeology points to a particular person starting the conversation off in September 2022 with messages like:

- our mascot should be an 8-bit horse
  Also the horse should be yellow
- Here's every horse: https://every.horse/
- Here's a horse that's shaking: https://shakingmy.horse/

The discussion significantly escalated on the next day, when the same person posted:

> Today I forked my personal link shortener to use it for Purdue Hackers. Introducing `puhack.horse`[2]
> ...

This person then served as the President of Purdue Hackers for three years, and led its rise from a 20 person meetup to a 80-100 person organization with sprawling projects, ideas, and coolness.

Is it any surprise that a club that emphasizes engaging in whimsy and creating things that bring joy would latch on to "horse"?

2. Putting the Special Interest Group in the Horse

We've established the importance of horses. Now, let's explore how the Special Interest Group part came about.

[1] Discord is an online group messaging platform used by Purdue Hackers. It is a centerpiece of the community, and many important discussions happen there.

[2] It redirects to https://www.purduehackers.com/

2.1. BURSTing from creativity

In Fall 2024, Purdue Hackers hosted a showcase for a bunch of projects that members had created. They called it BURST[3]. It was glorious. Seriously. Here're some photos from BURST to show you just how glorious it was. *I strongly encourage you to check out the website for more photos and information on the exhibits.*

BURST included (among other things) (in clockwise order):

- a phone bell whose insides had been replaced with a Raspberry Pi;
- the Purdue Hackers logo as a meter tall sign;
- a receipt printer; and
- an indie video game about running a boba shop.

[3]https://burst.purduehackers.com/

It was so glorious, in fact, that it challenged my imagination to think of it even *could* get more glorious. How could we ever top the projects that we'd showcased this year? How could we inspire more members of Hackers to make contributions to the next showcase we hosted?

How could *I* engage more members and spread the joy of creating and presenting?

2.2. Commit Overflow

For the past two years, Purdue Hackers has hosted the "Commit Overflow" event during Purdue University's winter break.

> Winter break is here; it's the perfect time to make the things you didn't have time to make this semester.
>
> During the last 10 days of the year, we're running Commit Overflow. The challenge: every day, commit to GitHub & post an update of what you're working on in #checkpoints[4]
>
> If you make it all 10 days, we'll send you stickers and a custom laser-cut badge that will never be made or distributed again.
>
> — The first Commit Overflow announcement in 2022

This event saw great participation from the community: people shipped commits and maintained a sense of connection over the break. I personally wrote a lot of documentation for keymashed[5], a project I'd showcased at BURST.

2.3. SIGBOVIK

Now to talk about something completely different: SIGBOVIK[6] (Special Interest Group in Harry Quark Bovik) is an yearly joke journal organized

[4] #checkpoints is a "channel" in Discord; a channel is a discrete subdivision within a server which members send messages to. #checkpoints, in particular, is a channel where people can showcase their in-progress creations.

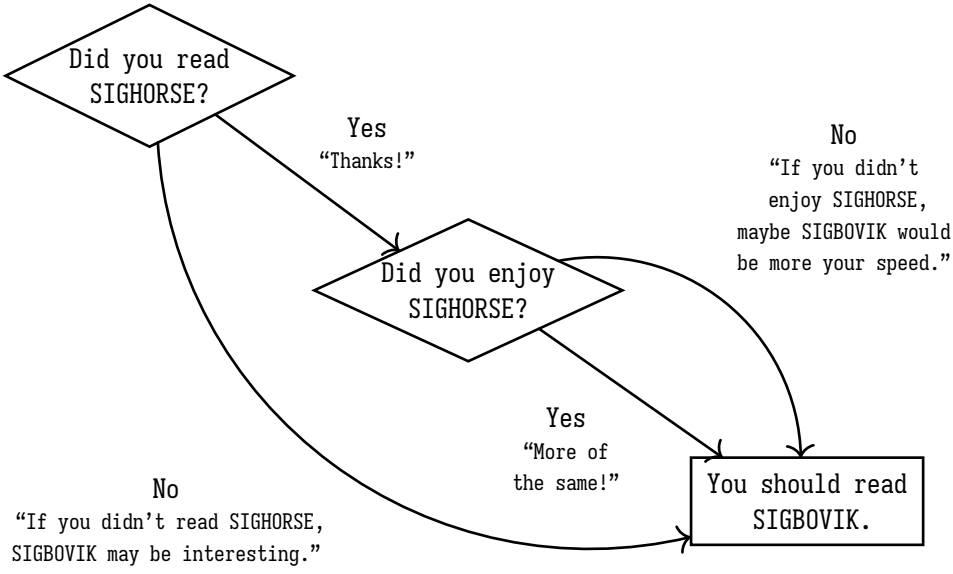
[5] https://github.com/kartva/keymashed

[6] https://sigbovik.org/

primarily by grad students from Carnegie Melon University with clearly too much time on their hands.

Their name plays on the Association for Computing Machinery's many conferences that start with SIG[7]: SIGPLAN (Special Interest Group on Programming Languages), SIGGRAPH (Special Interest Group on Graphics), SIGMICRO (Special Interest Group on Microarchitecture), etc.

The latest edition is just over 400 pages long. Papers published in this illustrious journal have had titles such as *"An Empirically Verified Lower Bound for The Number Of Empty Pages Allowed In a SIGBOVIK Paper"* or *"A Genius Solution: Applications of the Sprague-Grundy Theorem to Korean Reality TV"*.



[7]https://www.acm.org/special-interest-groups/alphabetical-listing

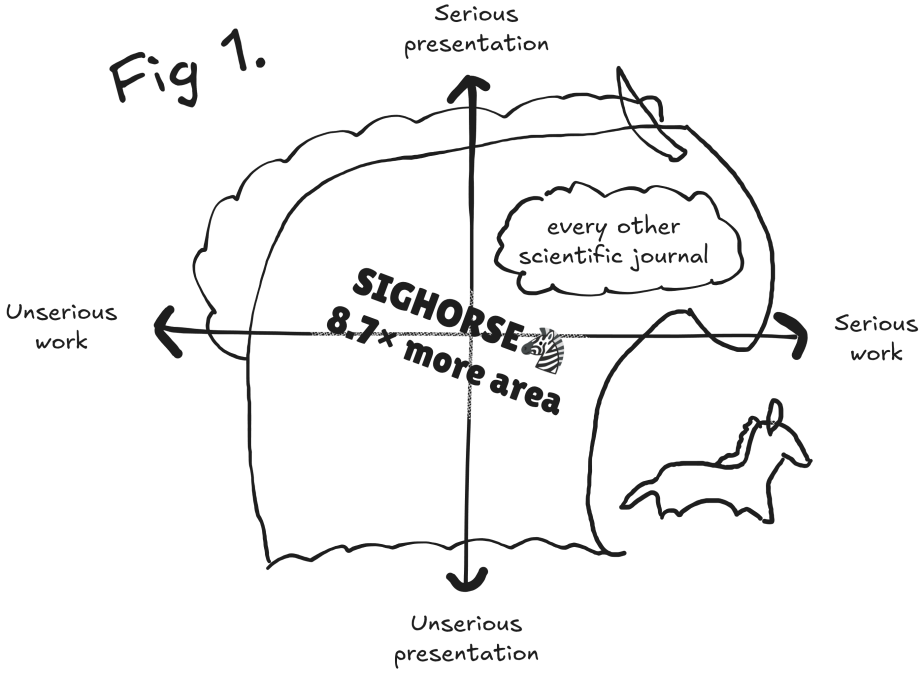
3. Messily pushing the horsing out to the world

3.1. Defining SIGHORSE

Finally, we can unify the two topics we discussed in the previous sections: SIGHORSE was proposed as "what if we ran something like Commit Overflow but in the summer and with a focus on whimsy and silliness?" Our tenets would be:

- encourage people to build cool things over the summer!
- talk about said things!
- get those things to a presentable state so we could put them in the journal!

To give an idea of the things SIGHORSE would cover, I produced the following diagram:



Simply put, SIGHORSE was to be inclusive.

(Before you ask, someone did measure the size of the horses and found it to be around 8.7× larger than the “every other scientific journal” cloud.)

3.2. Enough about SIGHORSE, what about the cover?

I’m so glad you asked about the cover! I first learned Blender[8], then Krita[9], then Inkscape[10], and then finally handed it off to an artist to finish because I sure couldn’t.

counterclockwise: first Blender draft, alternate Blender draft, Krita draft.
You can find the final version by looking at the cover.

[8] 3D modeling software: https://www.blender.org/
[9] Vector/Raster painting software: https://krita.org/en/
[10] vector drawing software: https://inkscape.org/

4. As the show curtains descend on the horse

SIGHORSE has been a blast. I hope you'll enjoy reading the submissions as much as the authors enjoyed creating them.


2048
meow

Teaching a Neural Network to Play 2048 (+ cat)

Angela Qian
Certified PHcker .ϟ
qian220@purdue.edu

Abstract

I decided to teach a neural network to play 2048, despite knowing very little about how to actually do that. This work represents a comprehensive summer-long case study employing the experimental methodology colloquially known as "fuck around and find out," formalized here as an iterative process of unstructured empirical exploration punctuated by intermittent bursts of ideas that appeared to me in my dreams. Despite training on a barely adequate dataset and a general disregard for best practices in machine learning, the project yielded a partially functional model that occasionally achieves non-embarrassing results. Future work will focus on replacing my uneducated, half-baked ideas with something vaguely resembling standard practice, as well as examining the effects of reading at least one relevant paper before implementation.

1 TL;DR: I made an AI that plays 2048

I've been mildly (okay, *extremely*) obsessed with 2048 since I was around ten years old. Funny tiles with big numbers itches my brain really good. Anyways, I'm studying computer science in college right now, and lately I've gotten interested in machine learning. So I figured, why not combine the two?

Figure 1: Author's artistic interpretation of this project[1] as a horse. Unclear if author has ever seen a horse.

2 Oops! I don't know what I'm doing

I got all hyped up about making a 2048 bot, but was quickly brought back to the crushing reality that: I'm a stupid little undergrad with a smooth little brain and do not know much about machine learning.

I started looking into some machine learning techniques, and the one that stood out to me most was imitation learning — basically, monkey see, monkey do. The reason? I honestly just wanted to avoid the headache of coming up with a way to "quantify" or "rank" how good a move is. With imitation learning, the model is given a bunch of state-action pairs (boards and their corresponding moves), and learns to predict the next action from a given state.

3 What would I do? Let's make the bot guess

Thanks to doing some undergrad research assistant stuff, I know that usually training these types of models requires *massive* amounts of data. As in, hundreds or even thousands of games. So of course, my first instinct is to search online for some pre-existing datasets I can use.

[1]This project can be found and played at https://angelazqian.github.io/2048-AI. Hopefully by the time you read this paper, my models will be slightly more competent than they were when I wrote this.

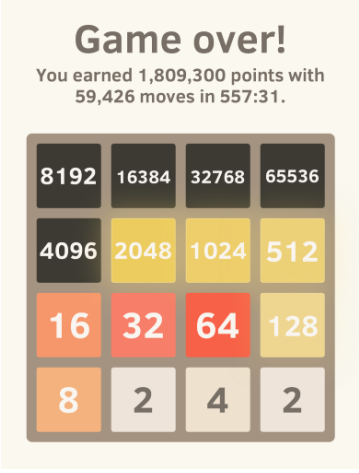


Figure 2: My highscore from last summer

I *did* manage to find some datasets of 2048 games online, but after digging into the stats, it turns out those players don't perform nearly as well as I do. For reference, Figure 2 shows my highscore from last summer.

Not only did their games tend to end much earlier than mine, looking through their gameplay, a lot of their moves were less than strategic. And as the saying goes — if you want something done right, do it yourself.

I ended up writing a small Python script that worked as a keylogger. When run, it opens 2048 in my browser, and for every move I make, the script saves the board state along with the move into a JSON file. This also allowed me to undo a move if I slipped up, so I didn't end up logging "bad moves" into the dataset. After collecting a staggering eight games (ok, not a lot lol but in my defense each game lasts anywhere from half an hour to four hours and i didn't have much free time since i was employed full-time over the summer), it's time to start training!

4 Forcing my model to see The Horrors

When looking into how to do imitation learning, the first method I came across was using Multilayer Perceptrons (MLPs).[2] Essentially, it's a type of neural network that processes multiple inputs, and returns a single output. This seemed like a good solution, as I could use the 16 grids of the game board as the input, then have it return the direction to move the tile in. I train it on the 8 games I have, load the model into the game and… yeah it sucks. Half the moves it was making were things I would never do. Back to the drawing board.

[2]MLP is also the acronym for "*My Little Pony*," which is rather fitting considering this is an entry in SIG**HORSE**. I thought this was hilarious so I have given my model a ponysona. See Figure 1.

Maybe the problem lies with my data? When I play 2048, I tend to shove my big tiles in the top-right corner, favoring the upper edge, as shown in Figure 3. This would be reflected in the dataset, since all the collected games would follow the same pattern.

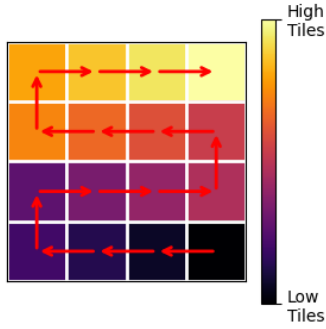


Figure 3: My preferred orientation

Since the model would have only learned to play well in the same orientation as me, when it loses that structure, it struggles to recover. That's also a problem if someone wanted to try playing the game themself, then swap in the bot mid-way — their tile alignment may be different than mine. Even natural gameplay sometimes shifts the board's orientation over time. I duplicated my data to represent all 8 orientations ($4 \times 90°$ rotations, then $2 \times$ for each mirror), as shown in Figure 4, and there is a *slight* improvement, but it's still comically bad.

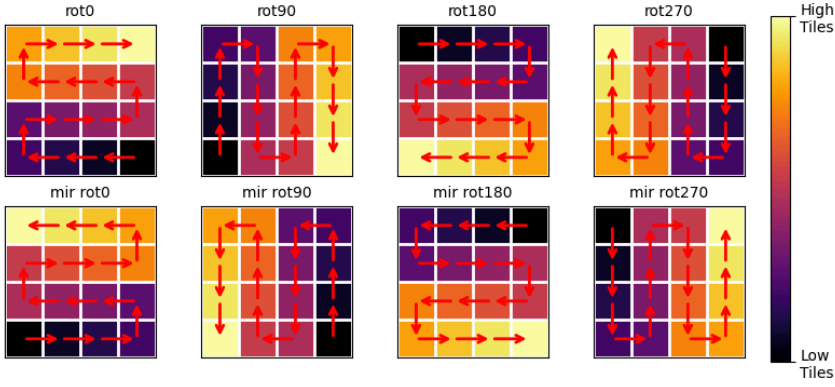


Figure 4: All 8 possible orientations of the board

After some thought, it occurred to me that treating each block in the grid as an independent parameter doesn't communicate any positional information, which is extremely important in 2048. To address this, instead of treating the grid as 16 independent parameters, I started treating the grid as an image, essentially implementing a rudimentary form of computer vision. I

accomplished this by switching from using MLPs to using a Convolutional Neural Network (CNN), which takes in a matrix input and uses convolutional layers to produce a single output. Much better results! But still not nearly as good as I had hoped.

5 Finetuning, but I am Woefully Uneducated

At this point I wanted to try finetuning my model, which means taking an existing trained model and continuing to train it so it becomes better adapted for the task. I looked into some common finetuning techniques, and the one that made the most sense to me was reinforcement learning through self-play, since 2048 is a single player game where you can objectively tell how good a game was through the final score.

In reinforcement learning, each full playthrough of the game is called an episode, and after each episode the model updates based on a reward function, which is basically a formula that tells the model what counts as "good". To encourage the model to be constantly improving instead of settling for an okay-ish score, I defined the reward function as the difference between the latest episode's score and the average score of past episodes.

And… it gets worse. What.[3]

I'll put this on the back burner for now and return to this later.

6 Making my model stop being Evil

The main issue about my model at this point is that it dies a lot early game, but if it somehow survives past a certain point, it starts performing well, which I think is because of rotation noise. This is because early on in the

[3] In the writing of this journal entry, I found out that my mistake was baking a moving baseline into the reward function, which makes reward non-stationary. What I was previously using as the reward function was actually something called the advantage (essentially how much better an action was than what the model usually expects). However, that should be handled inside the learning algorithm, not included in the reward itself. What I should have used here was a Deep Q-Network (DQN), which is designed to estimate long-term value for actions in each game state and updates the model more reliably.

game, the model seems to execute moves from various rotations, as if it can't decide which one to follow, leading me to remove the early gameplay from all rotations. I also noticed that when the board gains a large tile in a corner, it becomes ambiguous to the model as to which orientation it should follow. Considering that one of the main rule-of-thumb's when playing 2048 is that you should pick a direction, label it as "evil" and **avoid it at all costs**, this becomes a bit of a problem. For example, in Figure 5, if the largest tile is placed in the top-right corner, it *could* follow the orientation that favors the upper edge, designates `down` to be the "evil move", and mostly play moves `up`, `right`, and `left`. However, it could *also* follow the orientation that favors the right edge, designates `left` as the "evil move", and mostly play moves `right`, `down`, and `up`. Following this logic, all 4 directions may seem like reasonable moves to the model, which is *very bad*.

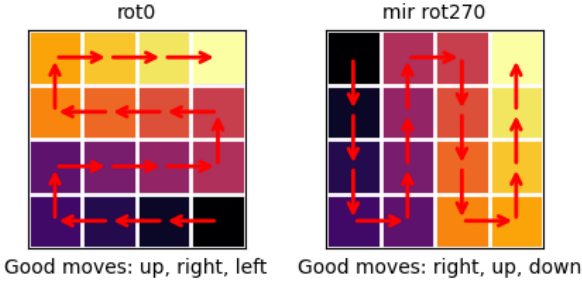


Figure 5: Ambiguity leads to all four directions being possible "good moves"

When playing 2048, it is good to choose one orientation and stick with it. However, sometimes a mistake happens, and you are forced to switch orientations in order to recover. The most common type of orientation change that happens mid-game is when you keep the same "evil move" and continue to favor the same edge, but switch to the other corner on the edge to keep the largest tiles. As an example, in Figure 6, the "evil move" continues to be `down` and the favored edge continues to be `up`, but the corner used to store the largest tile switches from the top right corner to being the top left.

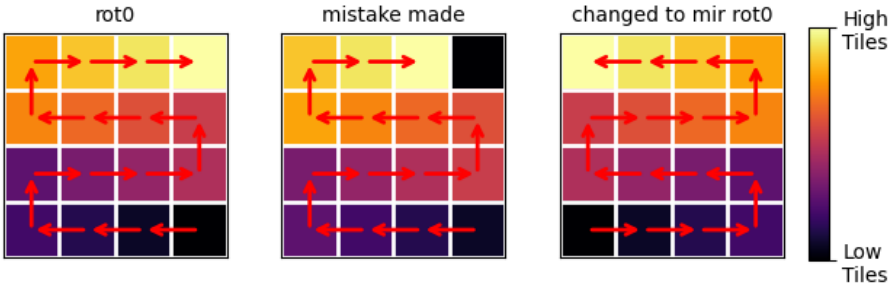


Figure 6: A common way of saving a game after a blunder is to switch orientations

To get rid of the orientation ambiguity for the model while still allowing it the flexibility to recover from blunders, I removed half of the rotations (the ones involving 90° and 270° rotations) from the training dataset. Just for good measure, I also removed all instances of when I was forced to do the "evil move" from all of the rotations. When I trained my model again on this new filtered data, I got much better results.

7 Cat

You're probably wondering where the cat comes in. There was "cat" in the title, you flipped to see what it was about, and instead got the deranged ramblings of some loser with an unhealthy obsession with 2048.

On July 18th around 10pm, my neighbor knocked on my door to tell me she heard what sounded like kitten meowing noises coming from my car. When I went to check, I could hear this little creature *wailing* from inside the car engine. I popped open the hood of my car, hoping to scoop him out, but the noise startled him, and he bolted into the surrounding bushes.

I regularly feed the neighborhood stray cats, and I know that all the strays in the area have been spayed or neutered, so the kitten had likely been separated from his mother. I couldn't bring myself to just leave him to the elements, so I sat on my porch with a bowl of Churu and unsalted chicken broth to try to lure him back out. He kept crying from the bushes and would occasionally dart

under other cars on the street, but he still wouldn't come near me. By 6am, I was cold, exhausted, and realizing this approach wasn't going to work.

The neighbor who first alerted me has experience trapping and rehabilitating stray cats. She's currently caring for an older cat and didn't want to risk exposing him to anything the kitten might carry, but she kindly lent me one of her humane cage traps. I put the Churu-broth bowl inside, set the trap, then went inside to rest for a bit.

When I checked a few hours later, the food was gone and I realized **he was too small to trigger the trap**. To fix that, I placed a 3 lb dumbbell on the pressure plate, refreshed the food, and waited. About an hour later — twenty-six hours after first hearing him — I finally caught him.

Figure 7: Tiny critter caught in a cartoonish cage trap

The next day, I brought him to a vet to make sure he was okay. They estimated he was about 8 weeks old and weighed only 1.59 pounds — on the low side for his age. He also had infections in both eyes and both ears, and was absolutely covered in fleas.

We started him on medication right away, and I kept him quarantined from my other cat while he recovered. After a few weeks of treatment, the vet gave us

the all-clear, and we finally introduced him to my resident cat, and thankfully, they hit it off. Have some pictures of them together.

Figure 8: Their first meeting!

Figure 10: They play with each other in a game almost like "cat and mouse." It's very entertaining to watch them chase each other at full speed.

Figure 9: My other cat started carrying him around the house by the scruff

Figure 11: The kitten has been imitating my big cat, including napping poses

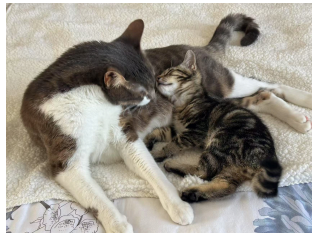


Figure 12: They like to sleep on each other all the time, using each other as pillows

Absolutely adorable. I love them both so much.

What was I doing before this again?

Oh. Right. 2048. Anyways, let's get back to it!

8 The model got too locked in

At this point, I had managed to collect about 25 games for training — much more than the 8 I started with, but still a tiny dataset compared to what most machine learning models thrive on. Around then, I started to suspect that my model was overfitting — in other words, it was getting too good at memorizing the training data instead of actually generalizing to new games. This is not ideal, because it means the model performs well on board states it has already "seen" during training, but struggles or completely fails when faced with new situations. To combat this, I added a dropout layer, which is a layer that will randomly "turn off" some neurons so the model becomes more robust and is less dependent on specific neurons. There's a lot of improvement!

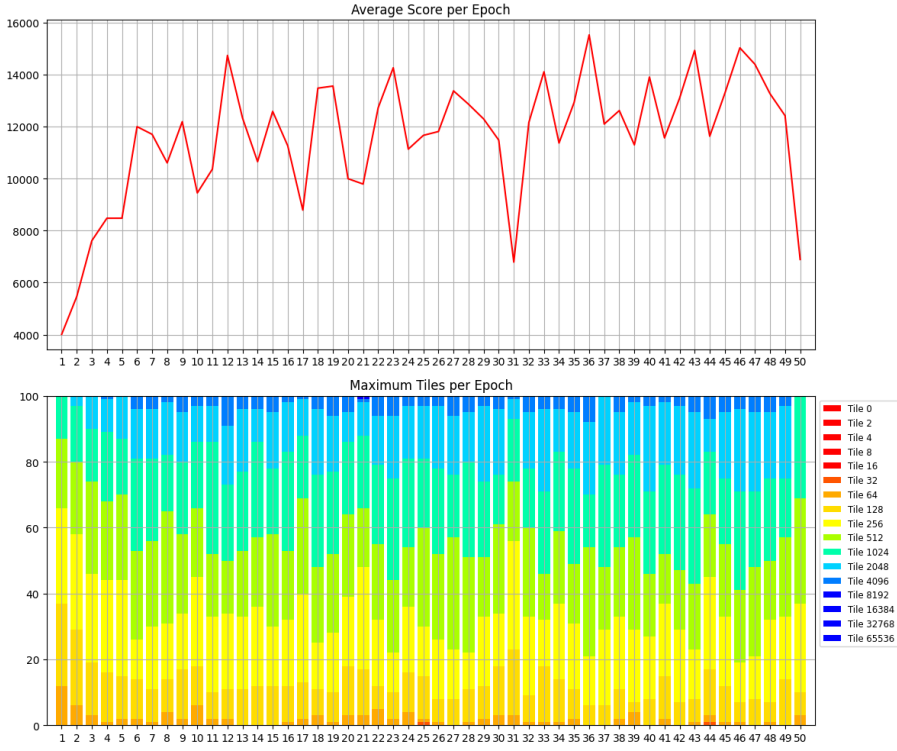


Figure 13: Performance of the model as epochs increase

However, I still felt that my model is overfitting. To monitor this, after every epoch (one full pass through the training dataset), I have the model play 100 games and record the average score as well as the distribution of the highest tiles reached. This way, I could track performance over time and identify when the model was actually improving versus just memorizing moves. At the end of training, I stored the weights from the epoch with the best average score, essentially picking the optimal epoch rather than blindly keeping the last one.

8.1 Brief interlude for the important graph that requires an entire subsection to explain properly

I've been told that the bottom graph in Figure 13 is a bit difficult to understand, so I'll try to break it down. Each bar shows the distribution of the highest tile reached during games played at that epoch. The proportion of a bar that's a given color corresponds to the proportion of games where the corresponding tile was the maximum. For example, at Epoch 36, around 7% of games ended with 128 as the max tile, while approximately 30% of games reached 2048 or higher.

Another way to read the graph is as a kind of "failure rate": the label on each bar tells you the percentage of games that failed to reach a certain tile. So for Epoch 36, the model fails to reach 512 about 21% of the time, and fails to reach 4096 92% of the time.

Here's the pseudocode for how the graph is generated, hopefully this helps if my explanation wasn't clear enough:

```
for each epoch:
    tile_distributions = {0, 0, 0,...}
    for each game in epoch:
        get max_tile created in game
        tile_distributions[max_tile] += 1
    for i in range(17):
        tile_value = 2^i
        show segment of length tile_distributions[tile_value],
            with color corresponding to tile_value
    show bar with these segments
```

8.2 Back to the main content, where I re-attempt finetuning

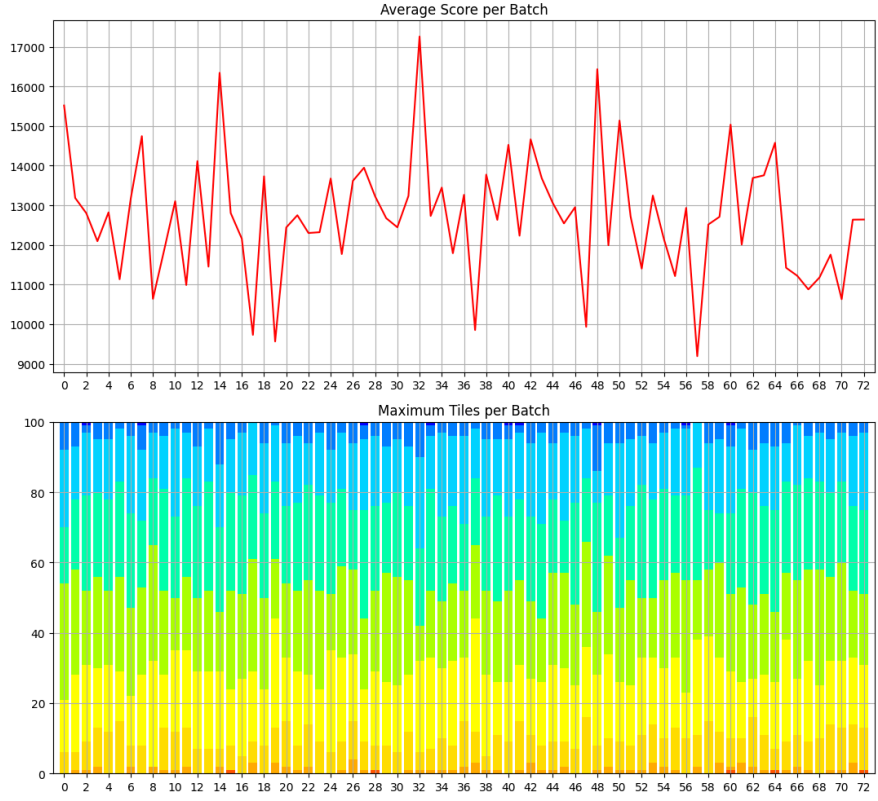


Figure 14: Performance of the model as batches increase

With the finetuning that I had attempted earlier, I noticed that the average scores reached by the model would dip a decent amount before they improved, then they would start worsening again. Because of this, I essentially did the same thing, where after each batch (a small group of training samples, in this case 25 episodes, processed before updating the model's weights), I test it for 100 games, then track the average scores. If no improvement is seen after 40 batches, I reload from the last best model, then continue from there. If it reloads

too many times in a row (8), I stop and end the training. Turns out, this actually does lead to some improvement!

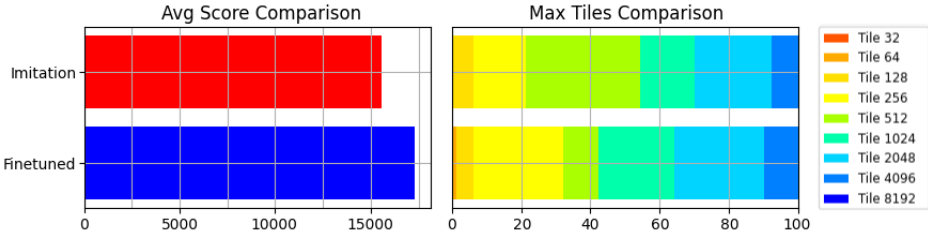


Figure 15: Comparison between pure imitation and finetuned imitation

9 The end?

My model is still far from perfect, and to be honest I'm still not completely satisfied with its performance. The finetuned version only reaches 2048 around 35% of the time, and since it was trained with imitation learning, it struggles to recover once it makes a mistake. It also does badly if you drop it mid-game where a human had been playing with a 90° or 270° rotation, since I excluded those orientations from my training data. Looking ahead, I'd like to experiment with models trained entirely through reinforcement learning without any of my own gameplay data, and eventually implement a DQN[4] into my model.

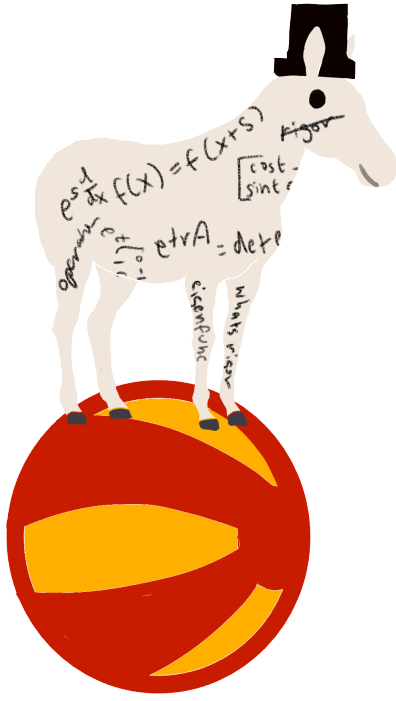
But alas, summer has come to an end, and with it, the end of my free time — and the end of my journal entry. Those are projects for another day.

This is my first ever journal entry, and thus, I have no idea how to end this. Bye bye, thanks for reading, etc. The end!!!

10 Acknowledgments

Thank you so much to Kartavya for helping me proofread and polish this. Also, huge thanks to Professor Campbell for double-checking my technical details and saving me from embarrassing myself. Lastly, thank you — you, the reader, for putting up with me long enough to reach the end of this paper.

[4]I explained what a DQN is in a previous footnote. Are you not reading my footnotes?! My feelings are hurt.


$\frac{d}{dx} f(x) = f(x+s)$
eigenfunc
whats nigga

Acromathics

Ishan Goel

Math classes are highly structured, and they just hand you results. It's way more fun to explore math on your own. I want to take you on three journeys, showing what it feels like to (re)invent math. The prereqs for this doc can change based on where you are, but not much prior knowledge is needed.

1 Roadside Gem

I'm in my AP Calculus AB class, and we've just learned about partial fraction decomposition. Here's a reminder of what that is: if you have a function that is the ratio of two polynomials, you can write it as a sum of simpler fractions. For example, $\frac{1}{x^2-5x+6} = \frac{1}{(x-2)(x-3)} = \frac{1}{x-2} - \frac{1}{x-3}$. Anyway, I'm facing this problem:

$$\int \frac{1}{x^2+1}\,\mathrm{d}x$$

… and I'm now a bit stuck, because I can't really factor x^2+1. Or… perhaps I can.

$$x^2+1 = x^2 - i^2 = (x-i)(x+i)$$

(By now, some of you might be screaming at the page about the integral being related to a certain trig function or whatever, but hey shh for now). Anyway, let's apply partial fraction decomposition:

$$\begin{aligned}
\frac{1}{x^2+1} &= \frac{A}{x-i} + \frac{B}{x+i} \\
1 &= A(x+i) + B(x-i) \\
\text{Let } \ x = i &\Rightarrow 1 = 2iA \\
\text{Let } \ x = -i &\Rightarrow 1 = -2iB \\
\therefore A &= \frac{1}{2i} = -\frac{i}{2} \\
\therefore B &= -\frac{1}{2i} = \frac{i}{2} \\
\frac{1}{x^2+1} &= \frac{i}{2}\left(\frac{1}{x+i} - \frac{1}{x-i}\right)
\end{aligned}$$

Now we can evaluate the integral:

$$\begin{aligned}
\int \frac{1}{x^2+1}\,\mathrm{d}x &= \frac{i}{2}\int \left(\frac{1}{x+i} - \frac{1}{x-i}\right)\,\mathrm{d}x \\
&= \frac{i}{2}(\ln|\ x+i\ | - \ln|\ x-i\ |) + C \\
&= \frac{i}{2}\ln\left|\frac{x+i}{x-i}\right| + C
\end{aligned}$$

You would be right to question this. What does it mean to take the natural log of a complex number? I have no clue. But hey, let's just assume this is valid and as a bit of a joke, you submit this wacky answer as homework and move on to the other problems. (Turns out you still get full points, but you suspect this is because your teacher does not look too closely)

Also, let's drop the absolute value signs. Like, at this point we're plugging in complex numbers, so negative numbers are the least of our worries.

$$\int \frac{1}{x^2+1}\,\mathrm{d}x = \frac{i}{2}\ln\left(\frac{x+i}{x-i}\right) + C$$

Later in class, I find out that the integral is actually a standard one and that:

$$\int \frac{1}{x^2+1}\,\mathrm{d}x = \arctan(x) + C$$

I didn't see this, and so now I had my own answer to the problem. Let's take a leap of faith and assume that my answer is valid. What happens if we equate the two answers?

$$\frac{i}{2}\ln\left(\frac{x+i}{x-i}\right) + C = \arctan(x)$$

(only one constant is needed)

Hmm, very interesting. Something involving logs and complex numbers on one side equals something involving inverse trig on the other. Maybe if we could find the inverse of this function, we could find a new way to represent $\tan(x)$. That would be interesting! But we need to find that constant C first.

(To be honest, at this point I put that expression into WolframAlpha to find what C is, but let's pretend I didn't do that and use a semi-rigorous argument instead.)

Since the two expressions are equal, their limits to infinity must be equal. Let's take the limit of both sides as $x \to \text{inf}$:

$$\lim_{x\to\infty}\left(\frac{i}{2}\ln\left(\frac{x+i}{x-i}\right) + C\right) = \lim_{x\to\infty}\arctan(x)$$

$$C + \frac{i}{2}\lim_{x\to\infty}\ln\left(\frac{x+i}{x-i}\right) = \frac{\pi}{2}$$

Hmm, we don't actually know what that limit on the LHS is, but let's make the argument that as $x \to \infty$, the difference in imaginary part "matters" less and less. So:

$$\begin{aligned} C + \frac{i}{2} \lim_{x\to\infty} \ln\left(\frac{x+i}{x-i}\right) &= \frac{\pi}{2} \\ C + \frac{i}{2} \lim_{x\to\infty} \ln\left(\frac{x}{x}\right) &= \frac{\pi}{2} \\ C + \frac{i}{2} \ln(1) &= \frac{\pi}{2} \\ C + 0 &= \frac{\pi}{2} \\ \therefore C &= \frac{\pi}{2} \end{aligned}$$

Finally, we now have a solid new representation for $\arctan(x)$:

$$\arctan(x) = \frac{i}{2} \ln\left(\frac{x+i}{x-i}\right) + \frac{\pi}{2}$$

Isn't that kinda cool? Yes we made some mildly shady arguments. But they're reasonable, and this is how discovery works. Come on, let's just see what happens. Let's try and find what $\tan$ is. Let's start by introducing two new variables:

$$\begin{aligned} \text{Define } \ u, v \ \text{ such that } \ \tan(u) &= v \\ \text{Then, } \ \arctan(v) &= u \\ \frac{i}{2} \ln\left(\frac{v+i}{v-i}\right) + \frac{\pi}{2} &= u \end{aligned}$$

If we isolate v in the above equation, we'll have a new representation for $\tan(x)$. Let's try:

$$
\begin{aligned}
u &= \frac{i}{2}\ln\left(\frac{v+i}{v-i}\right) + \frac{\pi}{2} \\
2u &= i\ln\left(\frac{v+i}{v-i}\right) + \pi \\
2u - \pi &= i\ln\left(\frac{v+i}{v-i}\right) \\
-i(2u-\pi) &= \ln\left(\frac{v+i}{v-i}\right) \\
(\pi - 2u)i &= \ln\left(\frac{v+i}{v-i}\right) \\
e^{i\pi - 2ui} &= \frac{v+i}{v-i}
\end{aligned}
$$

Hold on a sec. Do you see that? If only we didn't have that pesky $2ui$ we may be able to find out the value of $e^{i\pi}$! And that would be quite a gem.

Well let's try setting $u = 0$ and see what happens:

$$
\begin{aligned}
e^{i\pi - 2ui} &= \frac{v+i}{v-i} \\
\text{Set } \ u = 0 \Rightarrow e^{i\pi} &= \frac{v+i}{v-i}
\end{aligned}
$$

Welp. We don't really know what v is. So we can't find $e^{i\pi}$. Right? Wrong! We know what v is since we defined u and v to be related by $\tan$. Since $\tan(u) = v$, we know that when $u = 0$, $v = \tan(0) = 0$. So:

$$
\begin{aligned}
e^{i\pi} &= \frac{0+i}{0-i} \\
e^{i\pi} &= \frac{i}{-i} \\
e^{i\pi} &= -1
\end{aligned}
$$

And there, we have found the gem. But the road goes on, and so I strongly encourage you to carry on finding what $\tan$ is. It's a fun journey, and you rediscover Euler's formula among other things along the way.

2 Matrix Flow

This time we start in my third semester of college, in which I'm learning at the same time about both differential equations and linear algebra. Let me show you a cool link.

Let's start by considering an example from 3B1B. Suppose we have Romeo and Juliet, and two variables representing their love for each other, r and j. Let's say for some reason that Romeo loves Juliet more when she's being aloof, but Juliet's normal and likes Romeo more when he's being nice. We can represent this with a system of differential equations:

$$\begin{aligned} r' &= -j \text{ (Romeo's love grows when she's aloof)} \\ j' &= r \qquad \text{(Juliet's love grows when he's nice)} \end{aligned}$$

Where the $'$ symbol represents the derivative with respect to time t.

Let's first think about what we would expect the solution to this to look like. If Romeo loves Juliet more when she's aloof, and Juliet loves Romeo more when he's nice, then it seems like their love should oscillate. When Romeo is being nice, Juliet's love for him grows, but then he gets bored and becomes aloof, causing her love to decrease. This makes Romeo nice again, and the cycle continues. So we expect some sort of oscillatory solution.

Let's solve this system in whatever way we can. We realize that we can get a connection between the two equations by taking the derivative of either equation and substituting. Take the derivative of the first equation:

$$\begin{aligned} r'' &= -j' \\ \text{Substituting for } \; j' &= r \Rightarrow \\ r'' &= -r \end{aligned}$$

Aha! Now we're looking for a function whose second derivative is its negative. You might remember that our oscillatory friends sin and cos have this property. Let's try $r = \sin t$:

$$\begin{aligned} r &= \sin t \\ r'' &= -\sin t \\ j &= -r' = -\cos t \end{aligned}$$

But actually, $r = \cos t$ would work too. The full general solution, which you can verify, is:

$$\begin{aligned} r &= A\cos t + B\sin t \\ j &= A\sin t - B\cos t \end{aligned}$$

Where we choose A and B based on Romeo and Juliet's initial affections. You can verify $r'' = -r$ and $j = -r'$.

Let's find what A and B are if we are given r_0 and j_0, the initial affections at time $t = 0$.

$$\begin{aligned} r_0 &= A\cos 0 + B\sin 0 = A \\ j_0 &= A\sin 0 - B\cos 0 = -B \\ \therefore A &= r_0 \\ \therefore B &= -j_0 \\ \therefore r &= r_0\cos t - j_0\sin t \\ j &= r_0\sin t + j_0\cos t \end{aligned}$$

But you know, that felt a little unsystematic. What if we had some more complicated system?

We'll look at the same system in a different light in a second, but let's first take a detour to an unrelated* problem. Say we want to solve this really simple differential equation:

$$x' = x$$

What function is its own derivative? You might remember that $x = e^t$ works. Now let's consider a slightly more complicated problem:

$$x' = ax \qquad \text{Eq 1}$$

Take a second to check that this solution works:

$$x = e^{at} \qquad \text{Sol 1}$$

Let's now return to the original system, which I've copied here:

$$r' = -j$$
$$j' = r$$

Because we're trying to get a link to linear algebra, let's try to collect r and j into a vector x and try to write the system in the language of matrices and vectors.

$$\begin{bmatrix} r' \\ j' \end{bmatrix} = \begin{bmatrix} -j \\ r \end{bmatrix}$$
$$\text{Let} \quad x = \begin{bmatrix} r \\ j \end{bmatrix}$$
$$x' = \begin{bmatrix} r' \\ j' \end{bmatrix}$$
$$\therefore x' = \begin{bmatrix} -j \\ r \end{bmatrix}$$

Could we express that RHS in terms of x? It's already so close.

$$\begin{bmatrix} -j \\ r \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} r \\ j \end{bmatrix}$$
$$\therefore x' = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} x$$

Whoa! Take a second to appreciate what that last equation is saying. If we imagine x as a point in the 2D plane, this equation tells us that the velocity vector of that point is a 90 degree rotation of the position vector, naturally leading to a circular motion.

Already we see that this form gives us spatial intuition for the system. But can it help us further? Sometimes, to see further, we must see less. Let's name that matrix A and obscure its components. Now the system looks like this:

$$x' = Ax$$

Aha! Doesn't this look exactly like Eq 1 from before?! What if we could solve it in the same way?

$$\begin{aligned} x' &= Ax \\ x &\stackrel{?!}{=} e^{At} \end{aligned}$$

Well, is that it? Did we solve it? You should be flooded with questions. What does it mean to exponentiate At, a matrix? How can you multiply e by itself a matrix number of times? Obviously, this is all nonsense… right?

Let's create from scratch a way to exponentiate matrices.

First, do we even know what exponentiation is? Clearly, e^2 is just e multiplied by itself twice. But what is $e^{3.14}$? What is e^{π}? What is $e^{\sqrt{2}}$? You're somehow okay with these, but do you really know what they mean? You could argue that you can define exponentiation of the reals using successive rational approximations, but a much cleaner way is to use the Taylor series expansion of e^x, which we'll accept as fact.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

We see that this definition involves additions and multiplications, both of which we know how to do with matrices. So let's just use this definition to define matrix exponentiation. Let M be a matrix. Then:

$$e^M = 1 + M + \frac{M^2}{2!} + \frac{M^3}{3!} + \frac{M^4}{4!} + \cdots$$

But careful! For this equation to type-check, we'll replace 1 with the identity matrix I. Now we can *define* matrix exponentiation as:

$$\boxed{e^M \doteq I + M + \frac{M^2}{2!} + \frac{M^3}{3!} + \frac{M^4}{4!} + \cdots}$$

Now we can finally evaluate e^{At}. Let's try it out. First, let's compute a few powers of A.

$$\begin{aligned} A &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \\ A^2 &= \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \\ A^3 &= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \\ A^4 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \\ A^5 &= A \\ &\cdots \end{aligned}$$

Where we can see that A is just the 90 degree rotation matrix, and so its powers cycle every 4 applications. Now we can compute e^{At}:

$$\begin{aligned} e^{At} &= I + At + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \frac{(At)^4}{4!} + \cdots \\ &= I + At + \frac{A^2t^2}{2!} + \frac{A^3t^3}{3!} + \frac{A^4t^4}{4!} + \cdots \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + t\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} + \frac{t^2}{2!}\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} + \frac{t^3}{3!}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \frac{t^4}{4!}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \cdots \\ &= \begin{bmatrix} 1 - \frac{t^2}{2!} + \frac{t^4}{4!} - \cdots & -t + \frac{t^3}{3!} - \frac{t^5}{5!} + \cdots \\ t - \frac{t^3}{3!} + \frac{t^5}{5!} - \cdots & 1 - \frac{t^2}{2!} + \frac{t^4}{4!} - \cdots \end{bmatrix} \\ &= \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \end{aligned}$$

Where we recognized the Taylor series expansions of sin and cos in the last step.

Some of you may recognize this matrix as the rotation matrix that rotates a point by t radians. This makes sense, since we know the system should produce circular motion, but it also hints at a connection to Euler's formula, since $e^{it} = \cos t + i \sin t$ also represents a rotation in the complex plane, and i is a complex analog of our 90 degree rotation matrix A.

Finally, we can write down the solution to our original system:

$$\begin{aligned} x &= e^{At} \\ e^{At} &= \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \end{aligned}$$

Oh no! We know x is a vector, but our solution is a matrix!? Is all hope lost? I did think so for a couple days, until I realized that the complete solution to:

$$x' = ax$$

is actually

$$x = x_0 e^{at}$$

Where x_0 is the initial condition (value of x at $t = 0$). Perhaps we can fix our type issues by introducing a constant vector c storing the initial conditions:

$$\begin{aligned} x &= e^{At} c \\ \text{Let } \ c &= \begin{bmatrix} r_0 \\ j_0 \end{bmatrix} \\ \therefore x &= \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \begin{bmatrix} r_0 \\ j_0 \end{bmatrix} \\ &= \begin{bmatrix} r_0 \cos t - j_0 \sin t \\ r_0 \sin t + j_0 \cos t \end{bmatrix} \end{aligned}$$

Finally, our solution is:

$$r = r_0 \cos t - j_0 \sin t$$
$$j = r_0 \sin t + j_0 \cos t$$

Which exactly matches our previous solution! Hooray! While this seems more systematic, and it definitely has the seeds of generality, we still had the unsystematic part of the actual matrix exponentiation, where we relied on recognizing Taylor series. If we could just systematically exponentiate matrices, we could solve any system of linear differential equations!

Enter diagonalization!

Diagonalization is a method to factor a matrix into a product of three matrices such that the middle one is a diagonal matrix (zero everywhere except the main diagonal). The reason we care about diagonal matrices is because many computations, including raising them to powers, become trivial. For example, if D is a diagonal matrix:

$$D = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$

Then:

$$D^n = \begin{bmatrix} a^n & 0 & 0 \\ 0 & b^n & 0 \\ 0 & 0 & c^n \end{bmatrix}$$

Based on this property, we can see that the exponential of a diagonal matrix is also easy to compute:

$$
\begin{aligned}
e^D &= I + D + \frac{D^2}{2!} + \frac{D^3}{3!} + \frac{D^4}{4!} + \cdots \\
&= \begin{bmatrix} 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \cdots & 0 & 0 \\ 0 & 1 + b + \frac{b^2}{2!} + \frac{b^3}{3!} + \cdots & 0 \\ 0 & 0 & 1 + c + \frac{c^2}{2!} + \frac{c^3}{3!} + \cdots \end{bmatrix} \\
&= \begin{bmatrix} e^a & 0 & 0 \\ 0 & e^b & 0 \\ 0 & 0 & e^c \end{bmatrix}
\end{aligned}
$$

I like to think of diagonal matrices as having no interactions between different dimensions (no "cross-terms"), which is why we can simply exponentiate each dimension separately.

We would really like all our matrices to be as easy to exponentiate as diagonal matrices, thus motivating 'diagonalization'.

To build up to that concept, we first need intuition for eigenvalues and eigenvectors. An eigenvector of a matrix M is a vector that only gets scaled when multiplied by M. The amount it gets scaled by is called the eigenvalue. If v is an eigenvector of M with eigenvalue λ, then:

$$Mv = \lambda v$$

Why do we care about eigenvalues and eigenvectors? It's because this scaling property is super useful. For example, let's take this matrix:

$$M = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 4 & -1 \\ -2 & -4 & 4 \end{bmatrix}$$

It has the following eigenvalues and eigenvectors:

$$\lambda_1 = 2 \qquad v_1 = \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix}$$

$$\lambda_2 = 2 \qquad v_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\lambda_3 = 6 \qquad v_3 = \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}$$

Verify for yourself that $Mv_i = \lambda_i v_i$ for $i = 1, 2, 3$.

Because of this property, it's easy to evaluate the result of a vector being transformed by M if we can write it as a linear combination of the eigenvectors. Specifically, if:

$$x = av_1 + bv_2 + cv_3$$

Then:

$$\begin{aligned} Mx &= M(av_1 + bv_2 + cv_3) \\ &= aMv_1 + bMv_2 + cMv_3 \\ &= a\lambda_1 v_1 + b\lambda_2 v_2 + c\lambda_3 v_3 \\ &= a(2)v_1 + b(2)v_2 + c(6)v_3 \end{aligned}$$

This gives us a hint at how to diagonalize M. To evaluate Mx for any vector, we first need to break x down into its eigenvector components, scale each component independently, and then recombine them. We can express this process in matrix form. Let P be the matrix whose columns are the eigenvectors of M:

$$P = \begin{bmatrix} -2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & -2 \end{bmatrix}$$

Clearly, this matrix would do the recombination step (going from the scaled eigenvector components to the transformed vector). The scaling

step is done by a diagonal matrix D whose diagonal entries are the eigenvalues of M since each one happens independently. Now, we just need to come up with the matrix that breaks x down into its eigenvector components. This matrix is simply the inverse of P, denoted P^{-1}. The intuition is that P takes in eigenvector components and outputs the vector, so P^{-1} must take in the vector and output the eigenvector components.

$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

Putting it all together, we have:

$$Mx = PDP^{-1}x$$
$$\therefore M = PDP^{-1}$$

In this case:

$$P = \begin{bmatrix} -2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & -2 \end{bmatrix}$$
$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$
$$P^{-1} = \frac{1}{4}\begin{bmatrix} -1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & -1 \end{bmatrix}$$
$$M = PDP^{-1} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 4 & -1 \\ -2 & -4 & 4 \end{bmatrix}$$

This is a beautiful result. But is it useful? Well, let's see what happens when we try to exponentiate M if it has this factorization:

$$\begin{aligned}
e^M &= e^{PDP^{-1}} \\
&= I + PDP^{-1} + \frac{\left(PDP^{-1}\right)^2}{2!} + \frac{\left(PDP^{-1}\right)^3}{3!} + \frac{\left(PDP^{-1}\right)^4}{4!} + \cdots \\
&= I + PDP^{-1} + \frac{PD^2P^{-1}}{2!} + \frac{PD^3P^{-1}}{3!} + \frac{PD^4P^{-1}}{4!} + \cdots \\
&= P\left(I + D + \frac{D^2}{2!} + \frac{D^3}{3!} + \frac{D^4}{4!} + \cdots\right)P^{-1} \\
&= Pe^DP^{-1}
\end{aligned}$$

On the second step we used this result:

$$\left(PDP^{-1}\right)^n = PDP^{-1}PDP^{-1}\cdots PDP^{-1} = PDD\cdots DP^{-1} = PD^nP^{-1}$$

Awesome. If we can diagonalize M, all we have to do is replace the diagonal matrix D with its exponential, which is easy to compute.

It might seem that this operation only works for some matrices, but in fact, there's a sense in which a random matrix can almost always be diagonalized. Let's try diagonalizing A from earlier, and check if we again get the same result for e^{At}.

$$\begin{aligned}
A &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \\
\text{Eigenvalues:}\quad \lambda_1 &= i \qquad\qquad \lambda_2 = -i \\
\text{Eigenvectors:}\quad v_1 &= \begin{bmatrix} i \\ 1 \end{bmatrix} \qquad v_2 = \begin{bmatrix} -i \\ 1 \end{bmatrix} \\
P &= \begin{bmatrix} i & -i \\ 1 & 1 \end{bmatrix} \\
D &= \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix} \\
P^{-1} &= \frac{1}{2}\begin{bmatrix} -i & 1 \\ i & 1 \end{bmatrix}
\end{aligned}$$

Verify this factorization on WolframAlpha.[1]

Now we can compute e^{At}:

$$\begin{aligned}
e^{At} &= Pe^{Dt}P^{-1} \\
&= \begin{bmatrix} i & -i \\ 1 & 1 \end{bmatrix} \begin{bmatrix} e^{it} & 0 \\ 0 & e^{-it} \end{bmatrix} \frac{1}{2} \begin{bmatrix} -i & 1 \\ i & 1 \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} i & -i \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -ie^{it} & e^{it} \\ ie^{-it} & e^{-it} \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} e^{it} + e^{-it} & i(e^{it} - e^{-it}) \\ -i(e^{it} - e^{-it}) & e^{it} + e^{-it} \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} e^{it} + e^{-it} & \frac{-1}{i}(e^{it} - e^{-it}) \\ \frac{1}{i}(e^{it} - e^{-it}) & e^{it} + e^{-it} \end{bmatrix} \\
&= \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix}
\end{aligned}$$

Verify this on WolframAlpha.[2]

As expected, this gives us the 2D rotation matrix again! Now, let's recap our method for solving systems of linear differential equations powered by exponentiating matrices.

1. Write the system in matrix form $x' = Ax$
2. Diagonalize A into $A = PDP^{-1}$
3. Compute $e^{At} = Pe^{Dt}P^{-1}$
4. Write the solution as $x = e^{At}x_0$
5. Plug in the initial conditions directly into x_0 and the answer pops out.

[1] https://www.wolframalpha.com/input?i2d=true&i=%7B%7Bi%2C-i%7D%2C%7B1%2C1%7D%7D%7B%7Bi%2C0%7D%2C%7B0%2C-i%7D%7D%7B%7B-i%2C1%7D%2C%7Bi%2C1%7D%7D*0.5

[2] https://www.wolframalpha.com/input?i2d=true&i=simplify+%7B%7Bi%2C-i%7D%2C%7B1%2C1%7D%7D%7B%7BPower%5Be%2Cit%5D%2C0%7D%2C%7B0%2CPower%5Be%2C-it%5D%7D%7D%7B%7B-i%2C1%7D%2C%7Bi%2C1%7D%7D*0.5

We made two leaps of faith: first, we assumed that the solution for a single variable differential equation $x' = ax$ generalizes to the matrix case $x' = Ax$, and second, we assumed that matrix exponentiation is defined via the Taylor series expansion. We see that making reasonable choices in our leaps of faith lead to beautiful truths.

3 Taylor Might

I remember watching a 3Blue1Brown video that ended on the massive cliffhanger of what $e^{\frac{\mathrm{d}}{\mathrm{d}x}}$ is (in words, the exponential of the $\frac{\mathrm{d}}{\mathrm{d}x}$ operator), and so let's explore that.

To simplify things, let's use Heaviside's notation for the derivative operator.

$$D = \frac{\mathrm{d}}{\mathrm{d}x}$$

What's an operator, you ask? It's just something that takes in a function and spits out another function. For example, applying D (same as $\frac{\mathrm{d}}{\mathrm{d}x}$) to the function x^2 gives $2x$.

As usual, let's start by expanding e^D:

$$e^D = 1 + D + \frac{D^2}{2!} + \frac{D^3}{3!} + \frac{D^4}{4!} + \cdots$$

Does this maybe feel like an abuse of notation? Like, we're just using this Taylor series expansion in a way it's not meant to? It should. You could understandably scoff and say that this is complete nonsense. But again, let's just be reasonable whenever we run into problems, and see what happens.

Here's a bunch of questions you might reasonably ask (in order of decreasing obviousness)

1. What does multiplying two operators mean???
2. What's addition??
3. And what does multiplying by a scalar do?

Okay now let's come up with some reasonable answers.

1. Let's say that multiplying operators means applying them in sequence.

$$D^2 = DD = \frac{\mathrm{d}^2}{\mathrm{d}x^2}$$

2. Let's say that adding operators means applying them and adding the results.

$$\begin{aligned}(D^2 + 2D + 1)\sin x &= D^2 \sin x + 2D \sin x + \sin x \\ &= -\sin x + 2\cos x + \sin x \\ &= 2\cos x\end{aligned}$$

   Ah, but also $(D^2 + 2D + 1) = (D + 1)^2$, so let's confirm that gives us the same answer:

$$\begin{aligned}(D+1)^2 \sin x &= (D+1)(D+1)\sin x \\ &= (D+1)(\cos x + \sin x) \\ &= (-\sin x + \cos x) + (\cos x + \sin x) \\ &= 2\cos x\end{aligned}$$

   Wow, maybe we have some good stuff here.

3. I kind of already used it for the second one's answer. It's pretty obvious:

$$\begin{aligned}(1)f(x) &= f(x) \\ (\pi D)f(x) &= \pi(Df(x))\end{aligned}$$

Finally, armed with these reasonable definitions, we now know what this operation means. We don't yet know what it actually does, but we can evaluate it.

$$e^D = 1 + D + \frac{D^2}{2!} + \frac{D^3}{3!} + \frac{D^4}{4!} + \cdots$$

(Oh, btw, that 1 is an operator. Consider it the identity operator. Don't be fooled!)

Let's apply it to a couple functions and see what happens.

$$\begin{aligned}
e^D x &= (1)x + (D)x + \frac{(D^2)x}{2!} + \frac{(D^3)x}{3!} + \cdots \\
&= x + 1 + 0 + 0 + ... \\
&= x + 1 \\
e^D x^2 &= (1)x^2 + (D)x^2 + \frac{(D^2)x^2}{2!} + \frac{(D^3)x^2}{3!} + \cdots \\
&= x^2 + 2x + 1 + 0 + ... \\
&= (x+1)^2 \\
e^D e^x &= (1)e^x + (D)e^x + \frac{(D^2)e^x}{2!} + \frac{(D^3)e^x}{3!} + \cdots \\
&= e^x \cdot \left(1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots\right) \\
&= e^x e \\
&= e^{x+1}
\end{aligned}$$

Perhaps you are noticing a pattern by now… Let's try something a bit harder. Let's try $e^D \sin x$.

$$\begin{aligned}
e^D \sin x &= (1)\sin x + (D)\sin x + \frac{(D^2)\sin x}{2!} + \frac{(D^3)\sin x}{3!} + \cdots \\
&= \frac{\sin x}{0!} + \frac{\cos x}{1!} + \frac{-\sin x}{2!} + \frac{-\cos x}{3!} + \\
&\quad \frac{\sin x}{4!} + \frac{\cos x}{5!} + \frac{-\sin x}{6!} + \frac{-\cos x}{7!} + \cdots \\
&= \sin x\left(\frac{1}{0!} - \frac{1}{2!} + \frac{1}{4!} - \frac{1}{6!} + \cdots\right) + \cos x\left(\frac{1}{1!} - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \cdots\right)
\end{aligned}$$

Hmm, do those infinite sums look a bit familiar? Let's remind ourselves of the Taylor series expansions of sin and cos:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$
$$\therefore \sin 1 = \frac{1}{1!} - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \cdots$$
$$\therefore \cos 1 = \frac{1}{0!} - \frac{1}{2!} + \frac{1}{4!} - \frac{1}{6!} + \cdots$$

Thus:

$$\begin{aligned} e^D \sin x &= \sin x \cos 1 + \cos x \sin 1 \\ \sin(x+a) &= \sin x \cos a + \cos x \sin a \qquad \text{(angle sum)} \\ &\therefore \\ e^D \sin x &= \sin(x+1) \end{aligned}$$

Wow. Look at that simplification! That's crazy. Wow it really does seem like:

$$e^D f(x) = f(x+1)$$

Pretty odd. And it seems like Taylor series play a big role. Maybe we can figure out if this fact is generally true by assuming we have a series representation of some function $f(x)$ and then applying e^D to it. Let's try that.

Let's say that $f(x)$ has a certain series representation:

$$f(x) = a_0 + a_1 x + \frac{a_2}{2!}x^2 + \frac{a_3}{3!}x^3 + \cdots$$
$$f'(x) = a_1 + a_2 x + \frac{a_3}{2!}x^2 + \frac{a_4}{3!}x^3 + \cdots$$
$$f''(x) = a_2 + a_3 x + \frac{a_4}{2!}x^2 + \frac{a_5}{3!}x^3 + \cdots$$
$$\ldots \therefore f^{(k)}(x) = \sum_{n=0}^{\infty} \frac{a_{n+k}}{n!} x^n$$

Now let's apply e^D to $f(x)$:

$$\begin{aligned}
e^D f(x) &= f(x) + f'(x) + \frac{f''(x)}{2!} + \frac{f'''(x)}{3!} + \cdots \\
&= \sum_{n=0}^{\infty} \frac{a_n}{n!} x^n + \sum_{n=0}^{\infty} \frac{a_{n+1}}{n!} x^n + \frac{1}{2!} \sum_{n=0}^{\infty} \frac{a_{n+2}}{n!} x^n + \cdots \\
&= \sum_{n=0}^{\infty} \frac{x^n}{n!} \left(a_n + a_{n+1} + \frac{a_{n+2}}{2!} + \frac{a_{n+3}}{3!} + \cdots \right) \\
&= \sum_{n=0}^{\infty} \left(\frac{x^n}{n!} \sum_{k=0}^{\infty} \frac{a_{n+k}}{k!} \right)
\end{aligned}$$

Hmm, we don't really know what that innermost series is. I spent a couple minutes looking at it until I realized the following:

$$\begin{aligned}
\text{(As shown)} \quad f^{(k)}(x) &= \sum_{n=0}^{\infty} \frac{a_{n+k}}{n!} x^n \\
f^{(k)}(1) &= \sum_{n=0}^{\infty} \frac{a_{n+k}}{n!} (1)^n \\
&= \sum_{n=0}^{\infty} \frac{a_{n+k}}{n!}
\end{aligned}$$

This is almost the form we want. Now let's rename n to k and k to n to get this:

$$f^{(n)}(1) = \sum_{k=0}^{\infty} \frac{a_{k+n}}{k!}$$

Convince yourself that renaming is a valid move. Now we can substitute this into our expression for $e^D f(x)$:

$$\begin{aligned}
e^D f(x) &= \sum_{n=0}^{\infty} \left(\frac{x^n}{n!} \sum_{k=0}^{\infty} \frac{a_{n+k}}{k!} \right) \\
&= \sum_{n=0}^{\infty} \frac{x^n}{n!} f^{(n)}(1)
\end{aligned}$$

This is beginning to look a lot like a Taylor series expansion, except we seem to be missing the shift. Or are we?! Look:

$$\begin{aligned}\sum_{n=0}^{\infty} \frac{x^n}{n!} f^{(n)}(1) &= \sum_{n=0}^{\infty} \frac{(x+1-1)^n}{n!} f^{(n)}(1) \\ &= \sum_{n=0}^{\infty} \frac{((x+1)-1)^n}{n!} f^{(n)}(1) \\ &= f(x+1)\end{aligned}$$

That series perfectly matches the Taylor series expansion of $f(x+1)$ centered at 1. So it seems like we have a general result. If $f(x)$ has a series expansion with center 0, then:

$$e^D f(x) = f(x+1)$$

It's easy to see that we can extend this to any center c by constructing a new function $g(x) = f(x+c)$ and then applying the above result to $g(x)$. Thus, if $f(x)$ has a Taylor series expansion, applying the exponential of the derivative operator to it shifts the function by one!

A natural question to ask at this point is whether we can create any shift. For example, it's easy to see that applying e^D twice should shift $f(x)$ by 2. But also:

$$\begin{aligned}e^D e^D f(x) &= \left(e^D\right)^2 f(x) \\ f(x+2) &\overset{!?}{=} e^{2D} f(x)\end{aligned}$$

Huh. Based on this, we can see that it could be reasonable to conjecture that:

$$e^{sD} f(x) = f(x+s)$$

If we go through the earlier proof again but with s this time, it's not hard to see that:

$$e^{sD} f(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \left(\sum_{k=0}^{\infty} \frac{a_{n+k}}{k!} \cdot s^k \right)$$

Also not too hard to see that:

$$f^{(n)}(s) = \sum_{k=0}^{\infty} \frac{a_{k+n}}{k!} \cdot s^k$$

And so:

$$\begin{aligned} e^{sD} f(x) &= \sum_{n=0}^{\infty} \frac{((x+s)-s)^n}{n!} f^{(n)}(s) \\ &= f(x+s) \end{aligned}$$

Thus we have this general result. If $f(x)$ has a Taylor series expansion, then:

$$e^{s\frac{\mathrm{d}}{\mathrm{d}x}} f(x) = f(x+s)$$

Wow! Scaling the derivative operator by some number, then applying the exponential of that operator to a function, shifts the function by that number. Isn't that beautiful?

Why might this be useful? Great question. I guess, for example, you could derive the angle sum formula. Do let me know if you think of something. But also it's kind of just pretty.

More abuses of notation to consider:

$$e^{iD} f(x) \stackrel{?}{=} (\cos D + i \sin D) f(x) \stackrel{?}{=} f(x+i) \quad \text{Pretty sure this is fine}$$

$$(e^D)^{\int} f(x) \stackrel{?}{=} \left(e^{D\int}\right) f(x) \stackrel{?}{=} e f(x) \quad \text{Nonsense}$$

$$e^D e^{-D} f(x) = f(x) \quad \text{Definitely true}$$

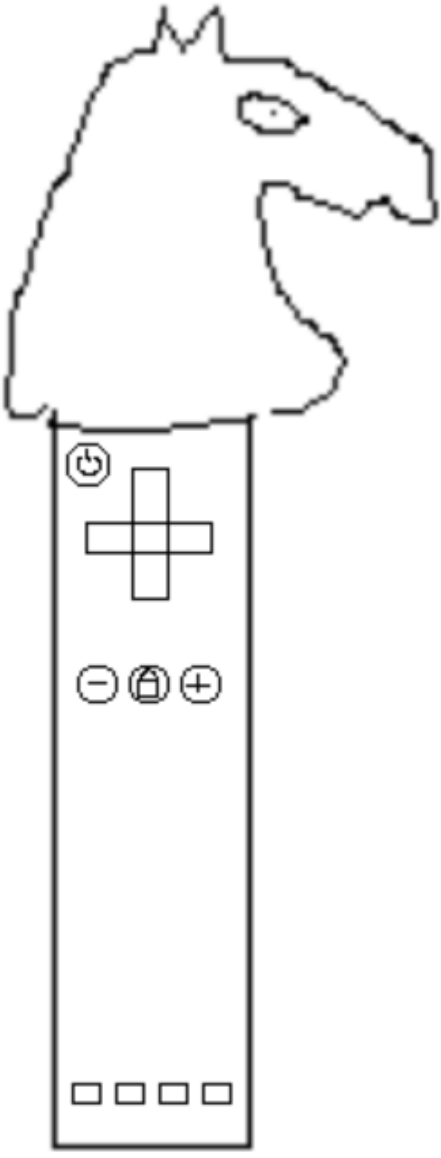
Try the first one! Another thing to try is to prove that the shift operator works for monomials without using Taylor series. You can then extend that to polynomials and then to all functions with series representations.

But I do want to remind you that notation abuse doesn't always work out:

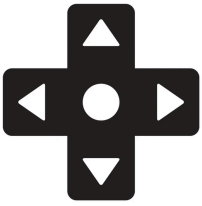
$$e^{i\pi} = e^{-i\pi} = -1$$
$$e^{i\pi D} \stackrel{?}{=} e^{-i\pi D}$$

One side shifts functions by $i\pi$ and the other by $-i\pi$, so it's definitely not true. (Do you see why it works in one case but not the other? Hint: inverse.)

I still think most people are too abuse-of-notation averse, so I encourage you to try that. Have fun and play responsibly!



How to NOT build a game controller in 10 easy steps



By Alexander Kutulas

December 11, 2025

Have you ever wanted to **NOT** build a game controller? Well don't worry; you've come to the right place! In just **10 EASY STEPS**, you'll be able to **NOT** build all the game controllers that you desire. And by all the game controllers, I mean just a single custom game controller called a Wii Baton, which is just a Wii Remote except with the A button, the 1 Button and the 2 Button replaced with a poor motion mapping system that is ONLY useful for playing Mario Kart Wii. So without further ado, "let's-a go!"

Figure 1: Picture of NO game controller

1. Design a 3D model without understanding ANYTHING about constraints

,

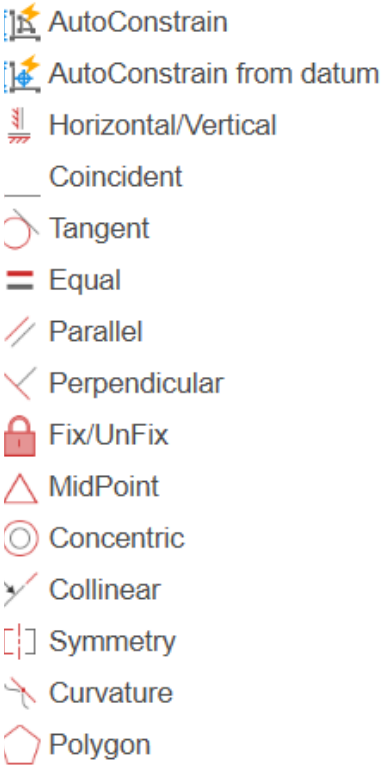


Figure 2: Constraints in Fusion 360

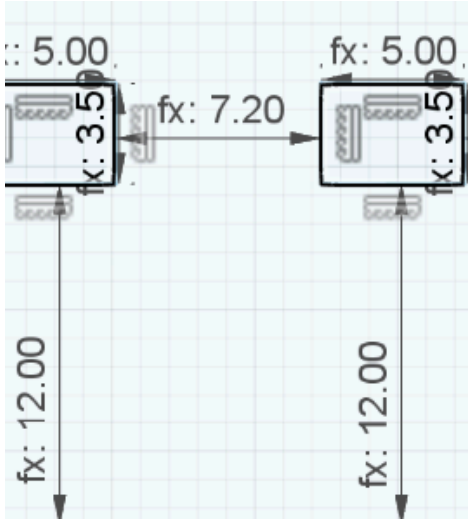


Figure 3: References in Fusion 360 (each 'fx' indicates a reference, which is why the dimensions are shared; changing the dimension being referred to will change all of the references)

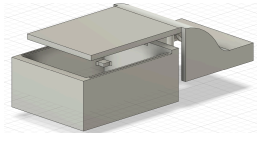


Figure 4: Trying to Frankenmesh separately designed parts together

1. Design a 3D model without understanding ANYTHING about constraints

First, you're going to want to design the body for your controller whilst thinking you're a **Fusion 360** (a 3D CAD, Computer Aid and Design software) hotshot, having designed approximately $3\frac{1}{2}$ models in the past. Completely ignore incredibly useful features such as **constraints** (Figure 2), which help with alignment and positioning, and **references** (Figure 3), where you can have dimensions equal one another – by selecting the dimension that you want to edit and then selecting the dimension that you want it to equal - because you don't know they exist. In particular, constraints such as:

- Coincident (constrains the position of two points together)
- Parallel
- Horizontal/Vertical
- Midpoint
- Symmetric

are incredibly useful, so make sure to avoid these! Finally, make sure to underuse 2D **sketches**; these are a core component of 3D design where you first design in 2D and then **extrude** your sketch outwards to add thickness, making it 3D. They can make your model more precise and easier to edit, and we wouldn't want that, so instead pretend you're using Blender and sculpt using the extrude and **push/pull** options in 3D mode to Frankenmesh your parts together (Figure 4) instead of making exact measurements in sketches.

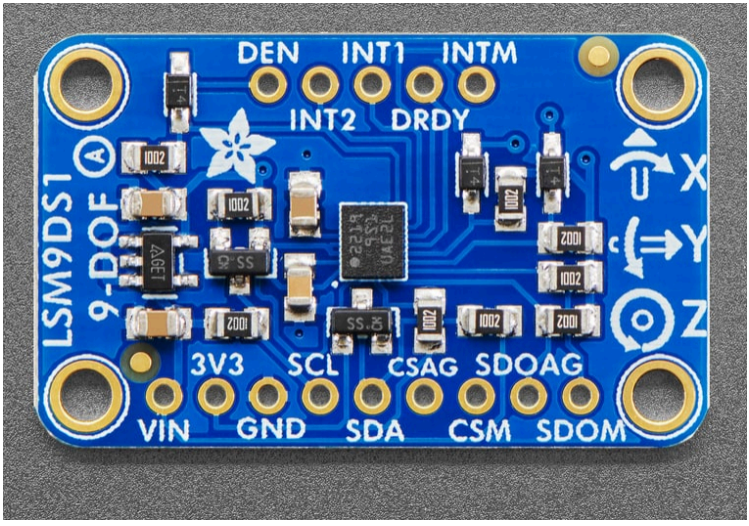


Figure 5: Adafruit LSM9DS1 9-DOF sensor

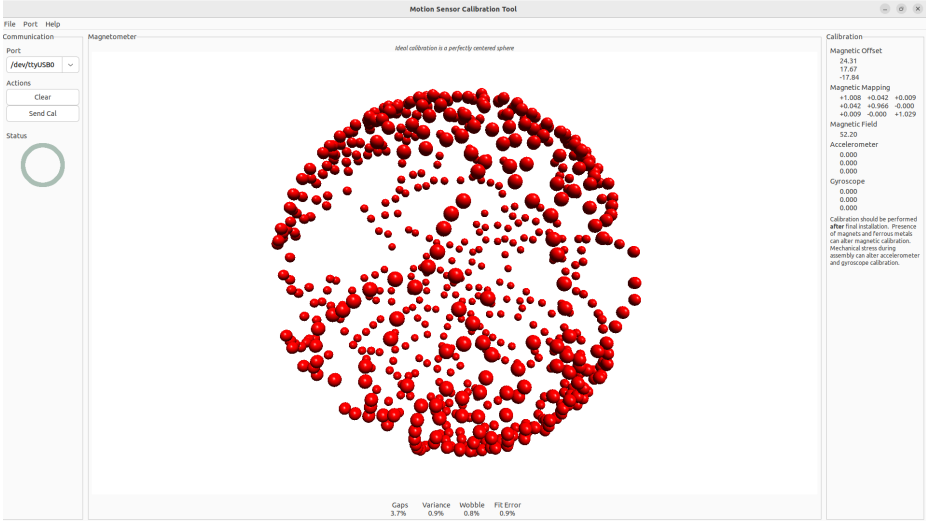


Figure 6: MotionCal sensor calibration software

2. Panic regarding sensor fusion

Next, assuming that you've already spent way too much time idly stripping wires while staring at a breadboard, you're going to NOT want to combine your 9 degrees of freedom of data (3 axis magnetometer, 3 axis gyroscope, 3 axis accelerometer) from your Adafruit LSM9DS1 sensor (Figure 5). Using the axes independently is much better, as the gyroscope drifts, the accelerometer is impacted by external accelerations like gravity and the magnetometer is susceptible to local magnetic disturbances, so if you don't:

- Represent the sensor's orientation using a **quaternion**, a 4D number like $q=w+xi+yj+zk$, where w represents the magnitude of the rotation and x, y, and z represent the axis of rotation
- Predict the new orientation using the gyroscope data
- Compare the prediction's gravity and magnetic north vectors to the measured gravity and magnetic north vectors. The difference is the **error**
- Adjust the quaternion β times to minimize the error (**gradient descent**)[1]

then your sensor data will be extremely inaccurate, thus impairing your ability to make a game controller using those sensors. And while we're at it, make sure to forget to calibrate your sensor! Definitely don't use incredible open source calibration software like MotionCal (Figure 6), that will take measurements as you rotate and move your sensor 9 different times (1 for each axis) and create a calibration sphere as you go, showing the values that your sensor is taking as a reference, as well as showing you where you might still need to calibrate.

[1]Explanation based on my understanding of a Madgwick filter; for more in-depth information read someone else's explanation, like Sebastian Madgwick's [1]

3. Implement a complicated finite state machine and subsequently take a month off

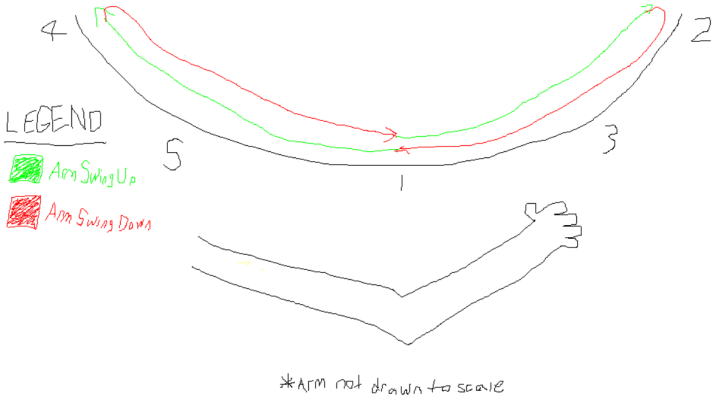


Figure 7: Diagram representative of the quality of my state machine code

3. Implement a complicated finite state machine and subsequently take a month off

Now that you've got a poor 3D model that you will most assuredly have to redesign and wildly inconsistent sensor data, it's time to write overly complicated state machine code[2]! For example, track the state of the motion of your arms while running and classify these motions between 5 different states:

1. IDLE
2. UP_FROM_IDLE_TO_UPPER_PEAK_FRONT
3. DOWN_FROM_UPPER_PEAK_FRONT_TO_LOWER_PEAK
4. UP_FROM_LOWER_PEAK_TO_UPPER_PEAK_BACK
5. DOWN_FROM_UPPER_PEAK_BACK_TO_LOWER_PEAK

and make sure to utilize lots of arbitrary constants to make the distinction between states, such as:

```
const float IDLE_ENTRY_MAG_THRESHOLD = 1.5f;
const float IDLE_EXIT_MAG_THRESHOLD = 2.5f;
const float SIGNIFICANT_DROP_FROM_PEAK = 1.0f;
const float SIGNIFICANT_RISE_FROM_VALLEY = 1.0f;
const float TREND_DETECTION_SENSITIVITY = 0.3f;
const int MIN_POINTS_FOR_PUMP = 10;
```

that will make it incredibly difficult and overwhelming to work with the codebase!

[2] I ended up scrapping this approach in favor of just normalizing and detecting motion in general for simplicities sake

4. Attempt to solder in the air because 'who needs PCBs anyway'

Figure 8: Home soldering setup

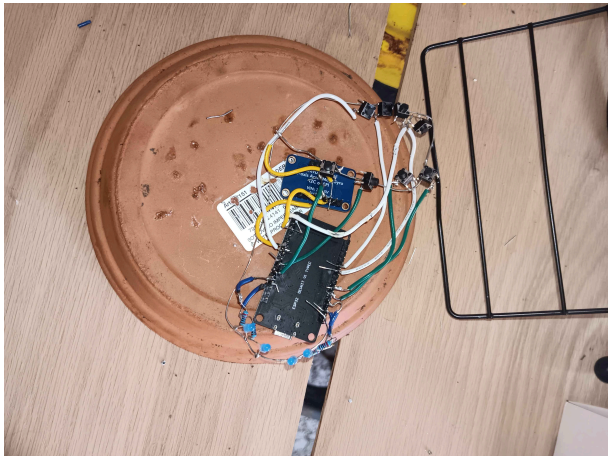


Figure 9: Well that could've gone better

4. Attempt to solder in the air because 'who needs PCBs anyway'

Now time to put it all together! Let's **solder** it all together by hand in the garage (Figure 8) on top of one of mom's ceramic pot lids that you found. We're using mom's ceramic pot lid as a base because soldering joins metallic connections by melting a filler metal (fittingly named 'solder'[3]), and ceramic, surprisingly enough, is not metal, meaning that the solder won't join itself to the base. We can also use mom's ceramic pot lid as a substitute for a **printable circuit board** (PCB).

A PCB functions as a great base for electronics projects, with tools such as **footprints**, **schematics** and the **PCB editor** that allow you to design a board to fit both your physical and electrical specifications. **Footprints** define the physical layout of electrical components; for example how big something is or how far apart **pads** (electrical connection points, often represented as a gold circle) on components are spaced. **Schematics** are where you design your electrical circuit, defining what is connected and where. Finally, **the PCB editor** is where you put it all together, placing and aligning your electrical components on your board. Different parts of the board will go on different **layers** of your PCB. Your pads will likely go on a **mask** layer, where solder mask will not be applied, allowing you to solder to your pads. The electrical connections that you made will take on the form of **traces**, copper paths that provide conductance between electrical components and which will be on a **Cu** (copper) layer. Any text or outlines will go on a **silkscreen** layer, which is essentially an ink coating that provides more information regarding your PCB. After you're done designing your PCB in an incredible open source software like **KiCad**, you can get it fabricated from a manufacturing company such as **JLCPCB** and then use it in your hardware projects!

All of this is exceedingly useful, so make sure to avoid designing and using a PCB and instead solder everything messily in the air with tons of wires so that it won't fit into your small 3D model! Components will shift around (Figure 9) and the single wire that you plan on using for ground gets really hot every time you try to solder to it and so you'll accidentally burn yourself a couple times, which should help discourage you from making a game controller.

[3]Note that solder does contain lead which can increase the risk of lung cancer if inhaled; if you plan to solder, I would recommend doing it in a well-ventilated space and use a fume extractor if you have one

5. Design the PCB too big because you extrude the 3D model body INWARDS instead of OUTWARDS

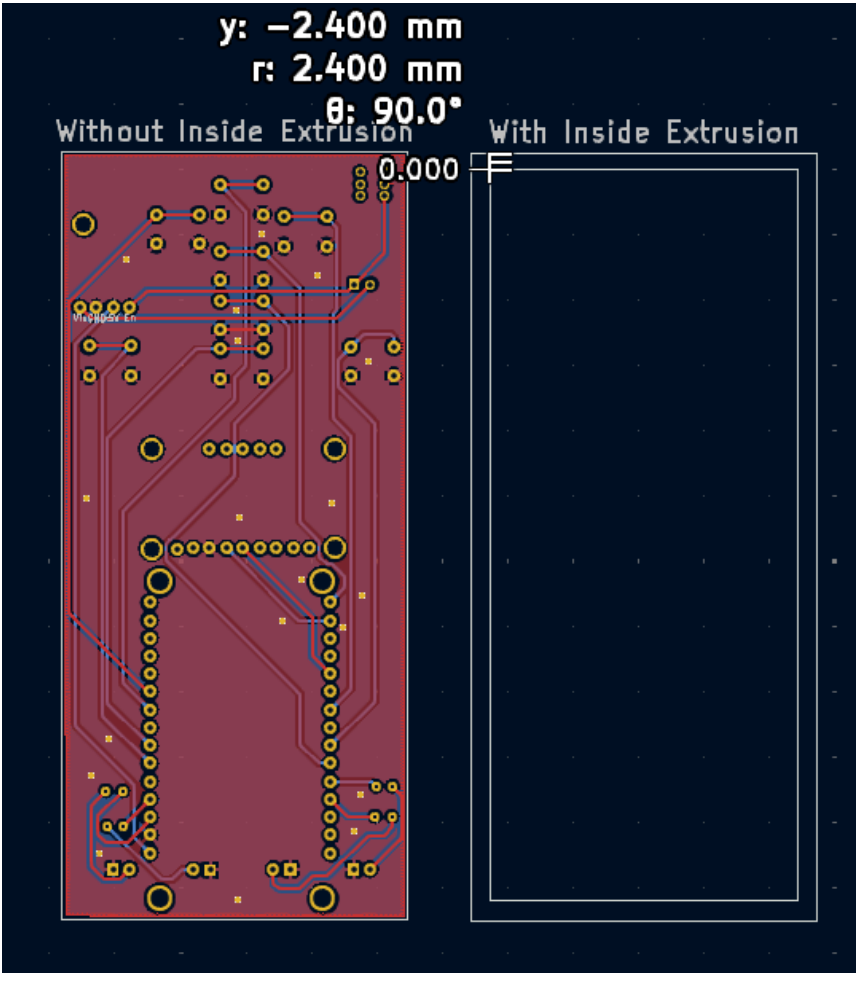


Figure 10: 2.4mm off? I'm sure it'll fit, you just gotta jam it in there

5. Design the PCB too big because you extrude the 3D model body INWARDS instead of OUTWARDS

If you still haven't failed to fail making a game controller at this point, you might've gone against my earlier advice and designed a PCB to make soldering easy and fit components inside your 3D model. In that case, don't export a DXF file from Fusion 360 and import it into KiCad via File > Import > Graphics to properly size your PCB, or, even if you do, make sure to forget that you extruded your 3D model INSIDE when adding thickness to the model. This will ensure that the PCB doesn't fit inside (fig 10), and successfully aid you along your quest to not build a game controller.

6. Space the holes for your ESP32 2.7mm instead of the breadboard-standard 2.54mm

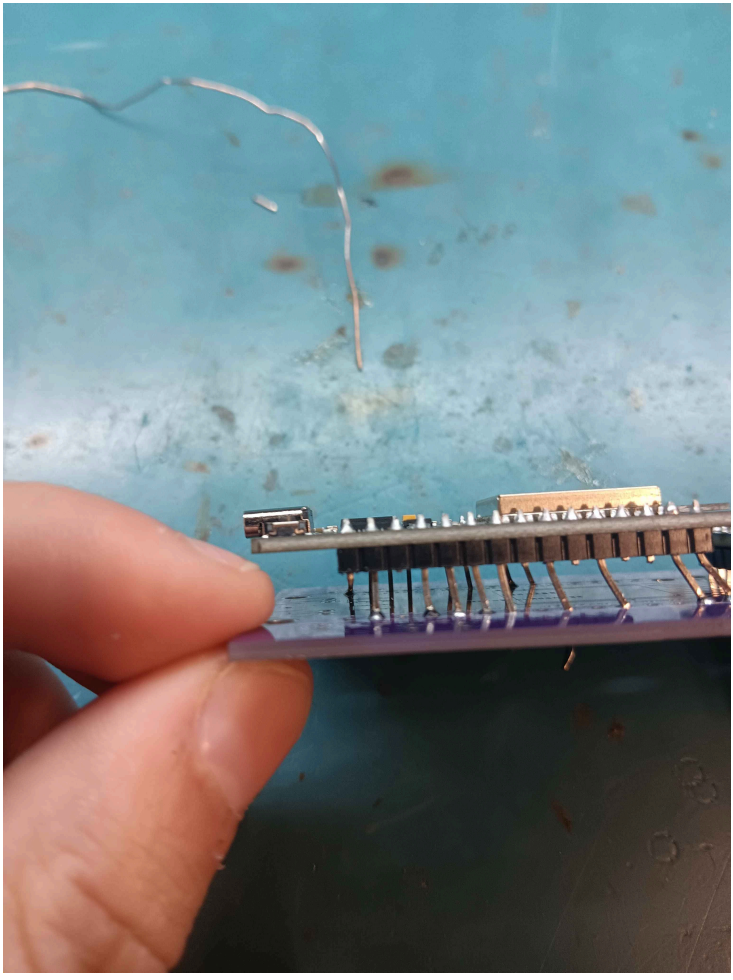


Figure 11: I prefer my pins at 45° angles

6. Space the holes for your ESP32 2.7mm instead of the breadboard-standard 2.54mm

Even if you decide to remake your 3D model to fit your PCBs, make sure that when you design your footprints that you find a way to ignore KiCad's grid and incorrectly space your pins for your **ESP32 microcontroller** – an **Integrated Circuit** (IC) that contains key modules such as Bluetooth and Wifi – by about 0.2mm. Definitely at this point you should succeed in failing to make a game controller and not try to bend the pins into the PCB (Figure 11).

7. Have your computer shut down when your circuit is connected via USB because there's a short

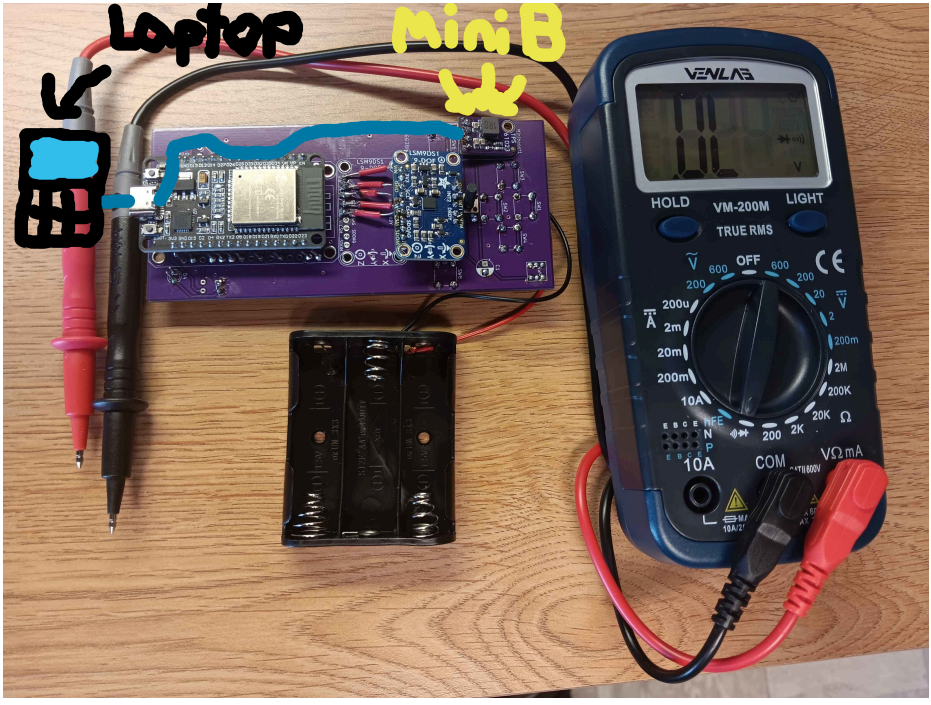


Figure 12: short

7. Have your computer shut down when your circuit is connected via USB because there's a short

Next on the agenda is to solder **GND** (the path of least resistance) and **5V** (POWER) together on your **Adafruit Miniboost** – another microcontroller to boost the voltage and current of a power input – and then NOT do a **continuity test** using your digital multimeter. Continuity tests are typically done to ensure that connection points like pins and pads are electrically connected, as well as to ensure that they AREN'T connected to each other. The multimeter in fig 12 is currently set to continuity mode; in this mode if you touch the ends of the probes to pins/components the multimeter will beep if they are connected[4]. This will ensure that all of the electrons rush out of your laptop to go party at Miniboost's place along the trace highlighted cerulean (Figure 12) and then cause your laptop to shutdown.

[4]DON'T do continuity tests while the circuit has power; this could damage your multimeter or possibly even the circuit itself. This is because during a continuity test the multimeter outputs a small current, and so if the circuit is live you can create a short depending on how you insert the probes into the circuit

8. Realize that bluetooth doesn't work because apparently RF zones on PCBs are a thing

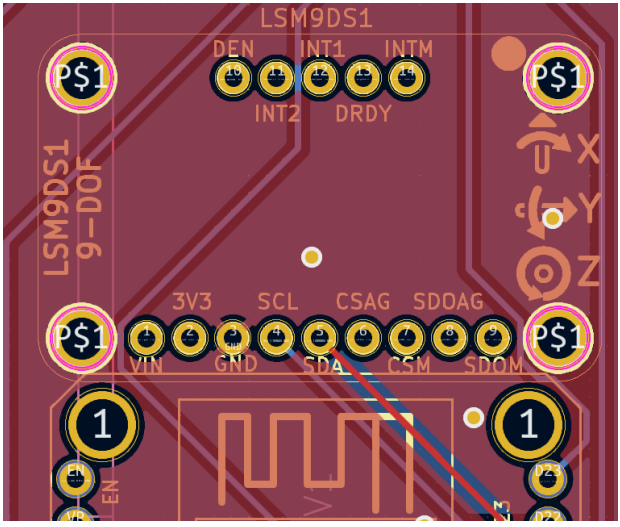


Figure 13: The ESP32 wants to be CLOSE friends with the GND vias and the LSM9DS1

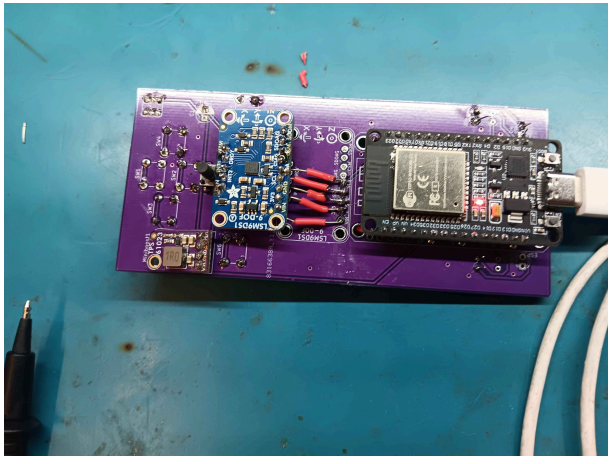


Figure 14: No more friendship; social distancing

8. Realize that bluetooth doesn't work because apparently RF zones on PCBs are a thing

Next, if you placed your ESP32 antenna (the square wave looking thing in Figure 13) near copper (e.g. ground vias) or other components like the LSM9DS1 so the antenna could have friends, you'll be in good shape. Copper placed in the path of the signal can cause impedance, reflecting part of the signal or causing it to take inefficient paths, and components like the LSM9DS1 can create additional noise when communicating using the **I2C** (Inter-Integrated Circuit) protocol. The I2C protocol utilizes 2 main lines (**buses**) of communication; the clock (**SCL** on Figure 13) and the data line (**SDA** on Figure 13). The clock is used to synchronize data transfers between the sender and receiver, and the data line contains the data and an ID to identify the sender. In the process though, if the clock is high speed enough it can create noise. All of these can interfere with the Bluetooth signal, meaning they'll help you be well on your way to NOT making a game controller. However, if you actually wanted to make a game controller, you might want to remove your LSM9DS1 from your PCB by giving it a quick buzz cut (mangling the pins) to get it out of the PCB and practice social distance with the ESP32 by moving them away from one another using wires (Figure 14), but we don't want to make game controllers, so there's no need to do that.

9. Break the MKWii physics engine by tampering with the speed

```
# C-stick X is used as a placeholder for speed and is set by the controller code
# Read the current value of C-Stick X (0x8034C202) into f31 and zero
lis     r12, 0x8034         # Load upper 16 bits into r12 (0x80340000)
ori     r12, r12, 0xC202    # Add lower 16 bits (now r12 = 0x8034C202)
li      r11, 0              # Clear r11
lbz     r11, 0(r12)         # Load 8-bit value into r11 (zero-extended)

# Convert the C-stick value (0-255) in r11 to a valid speed:
# 0 km/h = 0000, 5 km/h = 41C8, 50 km/h = 4248, 75 km/h = 4296, 100 km/h = 42C8...
# Output speed value will be stored back in r11

cmpwi   r11, 0
beq     set_base_speed_0      # if r11 = 0, set to 0 km/h

cmpwi   r11, 51
blt     set_base_speed_10     # 1-50 -> 10 km/h

cmpwi   r11, 77
blt     set_base_speed_25     # 51-76 -> 25 km/h

...

set_base_speed_0:
    li  r11, 0
    lis  r11, 0x0000      # 0 km/h
    b   set_base_speed

set_base_speed_10:
    li  r11, 0
    lis  r11, 0x4120      # 10 km/h
    b   set_base_speed

set_base_speed_25:
    li  r11, 0
    lis  r11, 0x41C8      # 25 km/h
    b   set_base_speed

...

# Use this value to set speeds (technically of the player and CPUs, but CPUs have a speed cap)
set_base_speed:
    # Inject > 80571CA4#

    # Set base speed
    mr r8, r11
    stw r8, 0x0018 (r3)
    lfs f1, 0x0018 (r3)

# Then follow the same process for max speed and inject at 80571CF4#
```

Figure 15: Welcome to the Wonderful World of Power PC Assembly

Figure 16: New YouTube Challenge Run: Can you beat Mario Kart Wii if you can't move from the start line and the game is rotated 180° for some reason?

9. Break the MKWii physics engine by tampering with the speed

If you somehow succeeded in getting the PCB assembled and functional, there's still plenty of software issues for you to fail at. For example, adjusting the speed of your character using Dolphin Emulators' built in cheat code system for the Gecko processor on the Wii. These **gecko codes** allow you to inject machine code into specified memory regions to execute during gameplay and are a great way to crash the game or cause an error state! Most memory addresses aren't safe to inject code into, but there are a few **hook addresses** where it is safe to inject machine code, so make sure to avoid hook addresses like **0x80571CA4**. If you inject your assembled PowerPC Assembly code ([2]) at **0x80571CA4**, the program will be in a state where you can edit the base speed of your kart by writing to memory (shoutouts to JoshuaMX for his code that does just that [3]). Don't worry though; you still have a chance to crash the game! If you write an 'invalid' speed value, you'll break the Mario Kart Wii Physics Engine (Figure 16), so make sure to avoid valid values such as:

- 0 km/h = 0x0000
- 5 km/h = 0x41C8
- 50 km/h = 0x4248
- 100 km/h = 0x42C8
- 150 km/h = 0x4316
- 200 km/h = 0x4348

Also make sure to avoid incredible tutorials such as Vega's *The Basics of Wii Cheat Codes & the Gecko Code Handler* [2]; these will describe the Gecko code system in incredible depth, discussing useful commands such as **C2XXXXXX 000000YY**, which will inject YY lines of assembly at memory address 80XXXXXX (80000000 is the start of the memory address block used in the Wii for games).

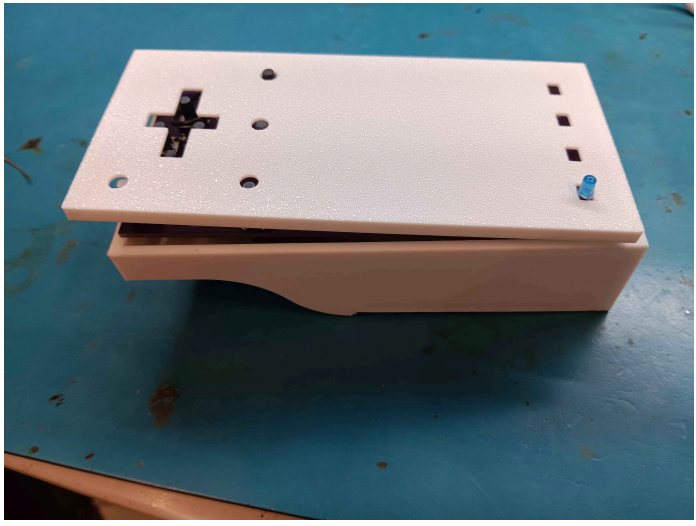


Figure 17: “It’ll fit if you try hard enough”

Figure 18: Soldering iron meets PLA

10. 'Fits' of frustration

If – by some manner of ungodly determination and will – you have made it this far without having succeeded in failing to make a game controller, then frustrations during the assembly process may aid you in your quest. Hopefully your tolerances will be slightly off, with buttons being automatically pressed by front covers and edges not lining up (Figure 17). Don't use your soldering iron to make the back button fit (Figure 18) or redesign your front cover to elongate the DPAD because you misplaced the buttons on the PCB.

Conclusion

By this point, I hope you've managed to fail in making a game controller. If, against all odds you have not, I'm truly sorry; perhaps throwing your completed controller against the wall may help? Otherwise, you might as well make the most of it and use your new controller to do a literal Mario Kart Wii speedrun.

After all, you've succeeded: in failing spectacularly.

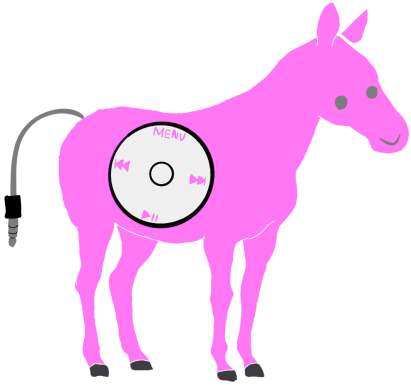
References

[1] Sebastian Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays." [Online]. Available: https://courses.cs.washington.edu/courses/cse466/14au/labs/l4/madgwick_internal_report.pdf

[2] Vega, "The Basics of Wii Cheat Codes & the Gecko Code Handler." [Online]. Available: https://mariokartwii.com/showthread.php?tid=434

[3] JoshuaMK, "Max Speed Modifier." [Online]. Available: https://mariokartwii.com/showthread.php?pid=3906

Special thanks to Grace for encouraging me to stick with the project and answering a ton of my questions along the way, to Hazel for the PCB Design advice and board review, and to Kartavya for reviewing this artifact and providing invaluable advice.


MENU

Self-Improvement, Habits, and iPods

The Tinkerer's Guide to Ditching Your Phone

Grace Yoder

I wouldn't say I'm addicted to my phone, but I also wouldn't say I have a healthy relationship with it. Like most people, I spend more time than I would like on my phone doing a variety of unhealthy behaviors. I spent so much time on my phone that at some point I turned off *Screen Time* tracking because I didn't like seeing how many hours I was spending on it. A typical day for me would typically include 7–10 hours of phone usage and over 100 pickups — that's about once every 10 minutes for all my waking hours.

Using my phone this much just feels gross. I didn't like the feeling that every waking moment of my life I was drawn to this magical computer slab in my pocket the same way a moth is drawn to light. Any moment of downtime would immediately be sucked up with a billion different apps and websites all trying to get me to stay there as long as possible. At the same time, I couldn't just get rid of my phone. My phone is a miracle device that can and does add significant value to my life. This device can connect me with people from across the world in real time, it knows where I am down to the meter, it can tell me exactly where to go when I get lost, it can capture a moment in time for me to look back at, it can let me into my building when I return home, it teaches me new ideas, and it allows me to learn more about the world

I am in. It does all of that while fitting in my hand. I can't just get rid of it, as my life (and modern life in general) is built around it, but I need to be healthier about it.

Now, I am not the first person to express these thoughts. There are many ways to go about reducing unhealthy habits while still keeping the utility of your phone. About a year ago, I met up for lunch with a friend that I hadn't seen for a while. At one point, he pulled out his phone and the way it looked shocked me. I asked him about it, and he showed me how he set his phone to black and white, gotten rid of all the apps on his home screen, and set a 15 minute limit on most trivial and unhealthy apps, all to help him reduce his dependence on his phone while still keeping it. I would consider to be drastic action, and many people are just able to set reasonable limits on their usage. I tried both reasonable limits and this more intense limit, but it wasn't enough for me. If I set it up, I could also bypass it, and having everything still be there is too tempting for me. I had to do something that would be harder for me to get around.

When *SIGHORSE* was announced, I decided to partake in a very ambitious project: fully getting rid of my phone. To do this, I would need to reorganize my life and replace all functionality of my phone with other devices. I think this is objectively one of the worst way to go about breaking habits, but I also knew this would allow me to see things in my life that I would otherwise be blind to.

Spoiler: I was not able to get to the point of leaving my phone at home at any point during this summer. However, I am prepared to do just that when the semester starts, and many of my bad habits have been changed or even fully broken.

So What Am I *Actually* Going to replace

There is a short list of the main useful functions of my phone:

- texting/general communication

- navigation in new places
- starting time tracking timers and managing to-do items
- key card to dorms and other campus buildings
- mobile hotspot
- recreationally watching YouTube
- listening to YouTube in the background while sleeping
- listening to podcasts while going between places
- camera (particularly for *Retro*)

These are all various levels of difficulty to solve for me. I can watch YouTube on my iPad. Texting/communication can all be done on my Mac or iPad, since I am (far too deeply) embedded in the Apple ecosystem. Navigation isn't an issue, since I am on a college campus and generally know where things are without maps. Also, Wi-Fi blankets campus so I do not need to worry about having a mobile hotspot. Timers, todos, and the keycard are solved with my Apple Watch connected to WiFi. Lastly, I will keep my phone in my dorm room hooked up to a charger most of the time, but will be able to use it at night so I can listen to video to help me fall asleep.

Most of these solutions are better than the alternative. Watching items on a bigger screen and messaging on a device with a real keyboard both benefit from me taking the time to take out a device better suited to the task.

This just leaves podcasts and a camera to be replaced. How hard could that possibly be?

Going Through iPod Hell

I will not talk about everything relating to iPods and podcasts here because the inane technical bullshit these devices put me through is mostly a distraction and irrelevant to the focus of this article. However, let it be known that I am not finished with these little electronic devils, and you can expect a longer piece about the technical side of iPods

and other DAPs in the future. For this, all you need to know is I started carrying two iPods:

- a 64 GB flash-modded iPod Mini running Rockbox with podcasts managed by my custom podcatcher called OxiPodder[1] that made it super easy to sync podcasts whenever I plugged it in
- an 8 GB 4th generation iPod Nano loaded up with my YouTube Music playlists.

Figure 1: My Flash-Modded iPod Mini

[1] https://github.com/gyoder/oxipodder

When they worked, I absolutely loved using my iPod for podcasts. iPods are actually nice objects and feel great to hold. Using the click-wheel to scroll through menus to select what I want to listen to and skipping forward or backward 15 seconds or pausing by clicking a real physical button is great and satisfying.

I would use it every day on my commute to work, when I would workout, and other moments during the day that I would always listen to a podcast. Having a separate device for podcasts removed the urge for me to change what I was listening to and play something else, or worse, start browsing YouTube. This gave me back the space that urge was taking in my mind and it feels calmer and clearer, even in a small way.

During this summer, I went on two road trips. The first one was with friends driving to Salt Lake City and Moab. I actually didn't use my player much on this trip and only listened to a single podcast. However, that wasn't because I was using my phone. Other than occasional navigation, I didn't use my phone at all. I think this was only possible due to having that urge to grab my phone removed, so at no point during the long ride did I even want to use it for entertainment.

My second road trip was to see family in Missouri. Due to *unforeseen accidents* (dropping an iPod at a gas station and falling off a jet ski with one I forgot to take out of my pocket), I had no iPod with podcasts for the way back, so I grabbed my phone and used that, since I was not about to go without anything for a 12-hour trip. What shocked me was the ripple effect this had on returning to all my old bad habits. The next morning, I found myself watching YouTube Shorts while eating breakfast before going to work. Additionally, that urge to grab my phone and change whatever I was listening to came back. Did they come back because I used my phone like I did before, or because I was a bit tired and it's easier to slip back into habits when you are mentally exhausted? I suspect a mixture of both, but it was a good reminder of what I was working to get rid of.

Figure 2: My iPod Mini needed a lot of repairs during its life

As much as I liked the iPods as a solution, they were not perfect. Neither the stock firmware or Rockbox are really designed for podcasts, so there are a lot of paper cuts. Additionally, it was a massive time sink for me to do this. I probably spent a minimum of 30 hours researching, modding, fixing, and debugging iPods and writing my own program to track and sync podcasts. Once I write a longer piece on that, I hope it won't be as difficult for others to do the same, but it won't ever be a fault-free solution. Also, it's expensive. I have probably spent upwards of $150 on all my iPod and parts. Despite all of this, when it worked, it really worked. I don't see myself without an iPod (or other Digital Audio Player) for a long time.

Replacing My Camera

At the start of this summer, I was roped into using an app called *Retro*. In a gist, *Retro* is a shared photo album only for friends. It does not feel like social media at all and really helped me connect with friends while I wasn't physically close to them. I never really got sucked in a death spiral on Retro since you can get to the end of it after 5 minutes. It also made me take at least one photo almost every day, which is great for documenting my life to look back on later.

This made it incredibly important for me to have a camera that time-stamps and geotags photos and was small enough for me to carry with me. Turns out, not really a thing in the modern day. Because phones have such good cameras, no one makes a modern point-and-shoot digital camera for about $100-200, since you can't beat the quality at that price point. There is a narrow range of cameras that have GPS in a small point-and-shoot form factor, but the used camera market is very expensive sadly. I found a few that looked good, but I already spent all my money on iPods so I was unable to justify buying one.

On my first road trip with my friends, I dusted off our family's old Nikon Coolpix point-and-shoot and took it with me. Notably, this was too big and didn't have GPS so didn't meet my requirements for a daily carry, but it was still nice to take photos with. The resolution doesn't look great, but I was able to get better photo composition using the optical zoom of the camera. I enjoyed it much more than phone photography, so I will likely shell out for a used point-and-shoot at some point in the near future.

Figure 3: Photo from Canyonlands National Park

Figure 4: Photo from Arches National Park

Figure 5: Photo of road from Canyonlands National Park

My current camera solution is another iPod. This time, it's an old 5th-gen iPod Touch found when cleaning out old tech. This is a shockingly good solution. Even though it is stuck on iOS 12, it is still able to sync all of its photos to iCloud (Apple does a surprisingly great job of supporting old hardware and software) and even includes GPS data. Additionally, it being stuck on iOS 12 is actually an advantage since there are almost no compatible apps (I have found zero) and the browser is so out of date that it can't load most modern single-page JavaScript apps. So far I have only gotten YouTube to work, but the YouTube mobile site sucks so much to use that this isn't an issue. This basically makes it a dedicated camera. The biggest downside is it's battery is shot, and it has about the quality that you would expect from a $200 device that was discontinued a decade ago. I will eventually get a real camera, but for now, I am happy with this.

Figure 6: iPod Touch 5th Generation

Nothing Quite Beats Pen and Paper

Phones are so good at everything that it is easy to be caught off guard if you don't have one on you. There are little things that you don't normally think about, like paying a friend on Venmo or needing to scan a QR code. While these do come up, they are mostly solved by another item I started to carry on me: a notebook. Rarely is anything that you need to do so urgent that it cannot wait until you finish the day and go back home. Because of this, you can write it down in a notebook and look at it later.

Additionally, I think that writing things out by hand, whether it be notes, logs, or ideas, is really beneficial. Handwriting helps with memory, and being able to mix in sketches, arrows, and everything else pen and paper allows for makes it easy to map out ideas in progress. It also just feels so nice. There is one very notable personal downside; My handwriting is historically bad and I often struggle to read what I've written down. However, this is balanced out by the memory advantages mentioned earlier. Ultimately, adding a notebook to my daily carry has been super nice.

Reflection

In Fall of last year, my first semester at Purdue University started. That person isn't me, or at least not who i am now. Ever since the first day that i have stepped on campus, my life has changed drastically. i have changed so much. i started working on super cool projects, i have met some really amazing people, i actually came out after 5 years. The life i have now is just significantly better than when i started and that doesn't seem to be changing any time soon.

Starting at the end of last semester, a new thought started echoing around in my brain.

> *oh god, i graduate in 2 years. i am almost a third of the way though my time here. i only get to do this again 2 more times. everything went so fast. if it continues at this rate ... but i cant leave. this place has made my life better. i cant leave the people i met. i cant leave everything ive built here. i love it here. what happens when i get out. i dont want my life stop getting better. it cant be over so soon. it cant finish before im ready...*

To this day, i still have this thought and it gets me really worked up and anxious.

...

Everything you have read up until this reflection section was written in August. It is now late September. I have written the conclusion many times and every time I hated it. It always felt like it was disingenuous, like I was trying to give you something that didn't exist.

A lot of the challenges that I faced when writing about this were the external deadlines. Because of the nature of *SIGHORSE*, I was working within a deadline so this project had to be over before it would naturally finish. I wasn't even able to really go out without my phone. Normally, something like this would be okay. If this was a story about technology, I wouldn't have had this issue. But it isn't a story about technology. It's a story that involves technology but is about me.

I tried to give some insight, some revelation, into my life and how I changed during this adventure. I wrote a cute little conclusion about how my life improved throughout this process and urged others to think about their life. It was an ending, but it wasn't great. It wasn't something that made a lot of sense in where I was in my life. My relationships to everything I wrote about was still changing

Then, something magical happened. *SIGHORSE* had review cycles. All of the sudden I was granted extra time and could write an ending that I liked more. I talked to some friends about the ending and wrote another one. This time I left the book open making sure to talk about how I still have a lot to learn about where this fits in my life and keeping it open for me to add to in the future. This just didn't feel satisfying. No mater how much I pretend otherwise, *SIGHORSE* is coming to an end and I need to acknowledge that.

...

During the darkest and brightest times in my life, i have always told myself "this too shall pass." i couldn't tell you where i first heard it, but it stuck with me. i was talking to a friend about my struggles with writing this ending. i told them about my insecurity about when college is over and they talked about how change was scary.

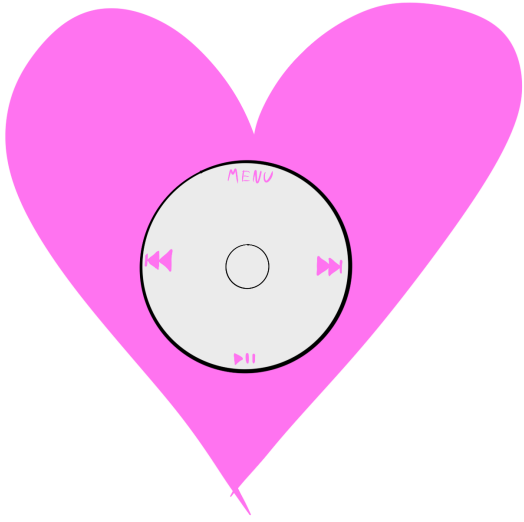
it surprised me when they said that, because thats not what i said. i said that i was scared that my life was going to stop getting better when i left. That i was going to lose everything i gained here. That i was scared of ... oh.

that i was scared of how my life was going to change when i was done.

...

Doing this adventure did make my life better and it would be silly for me to believe that once i submit this draft for *SIGHORSE*, that i would lose the personal progress i made. i will still have better habits and a cool new interest. it's also silly for me to think that everything i wrote about would stay static just because this project is over. i have another 70 years left in my life (optimistically anyway) so of course i will keep changing and of course my life will keep getting better throughout those years. just because one of the best chapters in my life will end, doesn't mean i won't find another one that will make me even better. it's a lot more satisfying to let something truly end and i need to be okay with it. i have a long time until i graduate so ill make the most of it, but when the end comes, ill be there in an open embrace to celebrate it and everything that's next.

this too may still pass, but i shall be out there, searching for what is after


MENU


"Hes"

Sarlacc: A Rust crate for lock-free interning of data

Henry Rovnyak

Abstract. In this report, I describe everything that I learned in the process of creating the Sarlacc crate in Rust, which is a lock-free implementation of interning. I describe the process and pitfalls of programming using atomics, as well as of creating a lock-free concurrent data structure. I discuss the fundamentals of atomic operations, memory ordering, and operating on pointers. Then I show how they apply to the Ctrie data structure and its implementation in the Sarlacc crate.

Introduction

Concurrency, the idea of doing multiple things at the same time, is an important feature of modern computer systems. For example, we can use it to speed up our programs by performing different aspects of a computation in parallel, or we can use it to improve the throughput and latency of our web servers by processing multiple requests concurrently.

Note that there are actually two kinds of concurrency: First, modern CPU chips are actually made up of multiple mini-computers wired together called *cores*, confusingly also called *CPUs*. Your operating system can put each of your concurrent operations, or *threads*, on different cores, allowing them to execute at literally the same time. This is called *parallelism*, which is a special case of concurrency.

However, if your operating system needs to run more threads than your computer has cores, it will be literally impossible to execute all of them in parallel — your operating system needs a mechanism to assign threads to cores such that all tasks get finished. This is done by instructing the cores to pause what they're doing and switch to a executing a new thread every few milliseconds. This is called *context switching*. It doesn't improve performance, but it allows all tasks to make progress, as opposed to letting some stall due to having to wait for other threads to finish.

From the perspective of a programmer however, we have no clue whether or not our threads will be executed truly in parallel or when or how often our threads will get stopped and restarted. Therefore, we have to engineer our code to function regardless of how our threads interleave in time and between CPU cores. This would be completely unproblematic if our tasks didn't depend on each other at all, but in real systems, threads will need some way to communicate and share information, requiring some form of *synchronization*. There are a lot of ways that this can go wrong.

The following code, where two threads increment a counter in parallel, illustrates a *race condition* — a situation where the output of a program depends on the unspecified order of execution of its concurrent threads of execution.

```
// Rust knows that what we're doing is
terrible; all of the unsafe is to stop the
compiler from trying to stop us
let mut number: u64 = 0;
let ptr = (&mut number) as *mut u64 as usize;

let handle = thread::spawn(move || {
    for _ in 0..1000000 {
        // Use volatile reads and writes to
prevent the compiler from collapsing them
into one operation
        let mut num = unsafe
{ read_volatile(ptr as *mut u64) };
        num += 1;
        unsafe {
            write_volatile(ptr as *mut u64,
```

```
num);
        }
    }
});

for _ in 0..1000000 {
    let mut num = unsafe { read_volatile(ptr
as *mut u64) };
    num += 1;
    unsafe {
        write_volatile(ptr as *mut u64, num);
    }
}

handle.join().unwrap();

// Almost guaranteed to NOT be 2000000
assert_ne!(number, 2000000);
```

This is a simple example, I swear 😅! Most of this is begging the compiler not to stop me from writing this horrible code and begging it to not optimize it to hide the issue. What is going on is that we're incrementing the same memory location over and over again from two different threads.

You can see that performing an addition requires three stages. A memory read, an addition, and a memory write. I separated those stages to prevent the compiler from collapsing the loop, but even if I just used a simple `num += 1`, the underlying implementation in the hardware would still require performing those three stages. The reason that the number printed is almost certainly not `2000000` is that it can happen that both the spawned thread and main thread can read, say, the number `1234` at the same time, perform the addition, and write `1235`. Two additions should have been performed, but it looks like only one has.

It may appear that operations that have only one stage are safe, for example maybe a single `write_volatile`. However, who knows how it's implemented in the CPU? It could be implemented in lots of different stages for all we know. Second, from the perspective of the CPU, it would be valid for the write to only be applied to a CPU core's local cache without being flushed to main memory. Then other threads wouldn't be able to observe the write which would cause all kinds of issues. I will elaborate more on this issue in later sections.

The classical resolution to this problem is to use a *lock* — a thing that will invoke CPU synchronization primitives to force threads to wait if they need to access data that is already being accessed by another thread.

```
// `Arc` allows us to share the `Mutex`
between threads without using unsafe code.
let number = Arc::new(Mutex::new(0_u64));

let number_for_thread = Arc::clone(&number);

let handle = thread::spawn(move || {
    for _ in 0..1000000 {
        // Locking the mutex prevents the
main thread from accessing the number until
the end of scope
        let mut num =
number_for_thread.lock().unwrap();
        *num += 1;
    }
});

for _ in 0..1000000 {
    let mut num = number.lock().unwrap();
    *num += 1;
}

handle.join().unwrap();

// Guaranteed to be `2000000`
assert_eq!(2000000, *number.lock().unwrap());
```

The problem with this is that we've immediately lost all benefits to using threads because threads have to *block* each other to be able to do work. When programming with locks, you have to think very hard to minimize the number of threads that need access to data behind a lock at any given time. The more subtle issue is that if the main thread gets interrupted while holding the lock, the new

thread will stall until the main thread gets rescheduled to release the lock.

Locks also have a lot of overhead because they typically involve invoking the operating system. This is necessary so that the CPU core can be rescheduled to do something useful while waiting, or simply get turned off to conserve power.

The solution that I'm building up to and the entire subject of this report is *atomics* — the aforementioned hardware synchronization primitive that allows us to build concurrent systems without blocking or invoking the operating system, and while preserving the benefits of parallelism.

Atomics effectively guarantee that an operation is performed in one stage; it's impossible for any CPU core to observe an intermediate stage of an atomic operation. This also implies that the atomic operation must immediately be visible to other threads and that it can't be hidden in the local cache of a CPU core.

```
let number = Arc::new(AtomicU64::new(0));

let number_for_thread = Arc::clone(&number);

let handle = thread::spawn(move || {
    for _ in 0..1000000 {
        // Atomically add one to the number.
        // Ignore the `Ordering` parameter, I
will talk about it later.
        number_for_thread.fetch_add(1,
Ordering::Relaxed);
    }
});

for _ in 0..1000000 {
    number.fetch_add(1, Ordering::Relaxed);
}

handle.join().unwrap();

// Atomically read the number
assert_eq!(2000000,
number.load(Ordering::Relaxed));
```

Finally, I can explain what my Sarlacc crate is supposed to be doing… Sarlacc uses atomic operations to implement a technique known as *interning*. Interning is when your program stores a piece of data in a global hash table forever such that all entries are deduplicated and effectively leaked from memory [1]. This has the benefit that a pointer to the data uniquely identifies it, and you can efficiently perform hashing and equality checks using the pointer instead of the (potentially very large) piece of data itself. The deduplication can also save memory depending on access patterns.

The reason that atomics are interesting for this usecase is that our hash table is *global* and accessible among all threads. Therefore, it needs to be synchronized somehow. Second, when we intern an object, we usually expect it to already be interned since we're not planning on leaking all of our memory. Therefore, the majority of operations will *read-only*. We don't need to enforce exclusive access to a piece of data if it's only ever being read, therefore using a lock to enforce exclusive access is overkill.

There does exist a type of lock called a `RwLock` that allows unlimited threads to access the data if they promise to only read, however we actually don't know whether our access will be read-only until we query the hash table and find out whether our value is already present in it. We could take a read lock and upgrade it to a write lock if we determine that we need to write, however ~~i hadn't thought of that when i started this project~~.

There is a ubiquitous crate for doing this known as `internment` [2]. As of writing, it implements the global hash table using a global lock, meaning it has all of the issues associated with locks that I describe above. This is the

motivation for re-implementing the functionality using atomics instead of locks.

In this report, I will explain to you, dear reader, everything that I have learned about atomics and how you can implement your own lock-free data structures from scratch. I will discuss the Ctrie data structure and its implementation in the Sarlacc crate, and I will discuss its performance.

Programming with Atomics

As our first lock-free data structure, we will re-implement `LazyLock` using atomics. `LazyLock` is a data structure intended for lazy initialization of global data.

```
static LAZY: LazyLock<u64> = LazyLock::new(||
{
    println!("Initialized!");

    123
});

println!("Starting!");
// Prints "Initialized!" because we are
accessing the data for the first time
println!("{}", &*LAZY);
// Does NOT print "Initialized!" because the
value that we initialized previously has been
saved
println!("{}", &*LAZY);
```

`LazyLock` uses a lock internally to prevent other threads from attempting to initialize the data while another thread is currently initializing it. Note that this is actually a very good usecase for a lock; it would be inefficient and potentially slower for all of the threads to try to initialize the data at once and to race to be the first. However this example demonstrates a lot of principles, so we will do it anyways.

Lets begin with a naive implementation, and call it `LazyAtomic` for funsies since it doesn't have locks.

```
pub struct LazyAtomic<T: Send + Sync> {
    // `null` will represent uninitialized.
    // `AtomicPtr` is essentially the same
thing as an `AtomicUsize` but it's a pointer
    data: AtomicPtr<T>,
    // This is the function we will call to
initialize the data once we have to do that
    initializer: fn() -> T,
}

impl<T: Send + Sync> LazyAtomic<T> {
    /// Create a new `LazyAtomic` with the
given initializer
    pub const fn new(initializer: fn() -> T)
-> LazyAtomic<T> {
        LazyAtomic {
            data:
AtomicPtr::new(ptr::null_mut()),
            initializer,
        }
    }
}

impl<T: Send + Sync> Deref for LazyAtomic<T>
{
    type Target = T;

    /// Either initialize or get the already
initialized value in the `LazyAtomic`.
    fn deref(&self) -> &Self::Target {
        let ptr =
self.data.load(Ordering::Relaxed);

        if !ptr.is_null() {
            return unsafe { &*ptr };
        }

        let initialized = (self.initializer)
();
        let initialized_ptr =
Box::into_raw(Box::new(initialized));

        self.data.store(initialized_ptr,
Ordering::Relaxed);

        unsafe { &*initialized_ptr }
    }
}

impl<T: Send + Sync> Drop for LazyAtomic<T> {
    /// Drop the LazyAtomic when it goes out
of scope. Rust doesn't drop raw pointers
automatically.
    fn drop(&mut self) {
        // We can access the pointer without
using atomics because we have a mutable
reference to it which guarantees that the
pointer is unaliased
        let ptr = self.data.get_mut();
```

```
        if !ptr.is_null() {
            drop(unsafe
{ Box::from_raw(*ptr) })
        }
    }
}
```

First, you'll probably notice that we have to use a decent amount of unsafe. This is just the reality of working with raw pointers in Rust. Second, you'll probably notice that there's a major problem with the atomic code in there! Don't read the next paragraph if you want to try to figure it out yourself.

What if two threads want to reference the data at the same time but it's not yet initialized? They will perform the load operation, and they will both read a null pointer. They will both call the initialization function and they will both try to store a pointer to their data into the `AtomicPtr`. The way that atomics work guarantees that one will do it after the other, but when the first one gets overwritten, the data is essentially leaked and lost forever.

To solve this, we need to use the *compare exchange* operation, also known as *compare and swap* or *CAS*. It is a very powerful atomic operation that has this functionality:

```
impl<T> AtomicPtr<T> {
    // This, but it's all atomic
    fn compare_exchange(&self, expected: *mut
T, new: *mut T) -> Result<*mut T, *mut T> {
        if self == expected {
            *self = new;
            Ok(expected)
        } else {
            Err(self)
        }
    }
}
```

This basically allows us to say "If the current value has not changed, then we can update it. Otherwise we can't." Lets see how we can use this to fix our `LazyAtomic` implementation:

```
fn deref(&self) -> &Self::Target {
    let ptr =
self.data.load(Ordering::Relaxed);

    if !ptr.is_null() {
        return unsafe { &*ptr };
    }

    let initialized = (self.initializer)();
    let initialized_ptr =
Box::into_raw(Box::new(initialized));

    match self.data.compare_exchange(
        ptr::null_mut(),
        initialized_ptr,
        Ordering::Relaxed,
        Ordering::Relaxed,
    ) {
        Ok(_) => {
            // Our value was successfully
inserted
            unsafe { &*initialized_ptr }
        }
        Err(prev) => {
            // Our value was NOT successfully
inserted; instead we found a different
pointer here which means that the value was
initialized after we loaded it.
            drop(unsafe
{ Box::from_raw(initialized_ptr) });
            unsafe { &*prev }
        }
    }
}
```

This actually works! It's a special case of a wider pattern that allows you to use compare-and-swap to make *any* operation atomic (with a caveat that I will explain later):

```
loop {
    let value = anything.load();

    let new_value = any_operation(value);

    match anything.compare_exchange(value,
new_value) {
        Ok(_) => break, // Update successful!
        Err(_) => continue, // Whoops, it
changed in the meantime! Lets try again.
    }
}
```

Such a "compare and swap loop" is the fundamental unit of operation for most lock-free data structures. Now this may look to you a

whole lot like a lock — when an operation is successful and overwrites the atomic pointer, it causes all other threads that are trying to update the value to fail. In effect, updates still happen in sequence. One of the benefits however is that reads are very lightweight: a simple atomic load. Lock free data structures shine in read-heavy workloads.

Another benefit is that if a thread working on the value is interrupted, it cannot cause the rest of the threads to block. Instead those threads will just keep going and the thread that got interrupted will probably have its CAS operation fail. In fact, this is the property that defines a *lock free* system. The term "lock free" does not refer to never using a lock explicitly — rather it refers to this property of guaranteed system-wide progress [3].

There is also a term, *wait free*, for systems that guarantee progress on every thread regardless of what other threads are doing [3]. A CAS loop is not wait free because progress on some threads can cause progress on others to stall potentially indefinitely. Read operations are typically wait free because progress on other threads cannot stall read operations.

There's actually yet another bug in our `LazyAtomic` implementation. I bet you won't get this one unless you know about...

Memory ordering

This section is essentially an amalgamation of the following sources [4] [5] [6] [7], and explained in the way that I wish it was explained to me. Memory ordering is a complicated topic and most sources seem to only give 50% of the explanation, though thankfully different 50%s. Hopefully I can give you 100%, though I encourage you to dive into those sources to strengthen your understanding.

What if I told you that even if a pointer was properly initialized, inserted into the `AtomicPtr`, and another thread loaded that pointer, that other thread might see uninitialized data? This may seem to break causality — how could that other thread possibly see the initialization and atomic store out of order?

Well, there are two reasons. The simplest one is instruction reordering by either the compiler or the CPU. For example, doing following reordering would be 100% fair game from the perspective of both the compiler and the CPU, assuming that they knew that the initializer was a pure function:

```
fn deref(&self) -> &Self::Target {
    let ptr =
self.data.load(Ordering::Relaxed);

    if !ptr.is_null() {
        return unsafe { &*ptr };
    }

    let initialized_ptr =
Box::into_raw(Box::new_uninit());

    let ptr = match
self.data.compare_exchange(
        ptr::null_mut(),
        initialized_ptr,
        Ordering::Relaxed,
        Ordering::Relaxed,
    ) {
        Ok(_) => unsafe
{ &*initialized_ptr },
        Err(prev) => {
            drop(unsafe
{ Box::from_raw(initialized_ptr) });
            unsafe { &*prev }
        },
    };

    *initialized_ptr = (self.initializer)();

    ptr
}
```

In fact, it would almost certainly be faster since you skip initialization in the failure case. This code is obviously incorrect, but the compiler doesn't understand the context in which

it is operating. It is clear how this would produce the effect that I described earlier.

The other potential cause of this issue is your CPU cache. I alluded to earlier how data being hidden in a cache local to a CPU core can prevent it from being seen by other threads. It is in fact possible for this to happen here. If the atomic access is observed but the initialized data isn't flushed into main memory or at least a higher level of cache, it cannot be observed by other threads.

Even if it is flushed out of the core-local cache, the thread accessing the data may need to flush its *own* cache to get the new data into it.

This is in fact what the `Ordering` parameters to the atomic operations are meant to solve. They explain to the compiler how the given atomic operation relates to other memory accesses on the same thread.

I discussed practical considerations like reordering and cache to help your intuition, but there is in fact a memory model describing exactly what guarantees the compiler must provide and which ones it doesn't have to. Fun fact, Rust actually uses the same memory model as C++.

The memory model defines five different types of ordering:

```
pub enum Ordering {
    Relaxed,
    Release,
    Acquire,
    AcqRel,
    SeqCst,
}
```

Relaxed ordering

This tells the compiler that your atomic operation has no relation whatsoever to other memory accesses. The CPU and compiler are free to reorder your code in any way —as long as it would be unobservable in a single-threaded context—and the CPU will not attempt to synchronize any memory other than that of the atomic itself.

Even though the ordering is relaxed, all threads must observe the same modification order for just that one atomic memory location.

For example, if one thread is executing function `a`,

```
static X: AtomicU64 = AtomicU64::new(0);

fn a() {
    X.fetch_add(5, Relaxed);
    X.fetch_add(10, Relaxed);
}

fn b() {
    let a = X.load(Relaxed);
    let b = X.load(Relaxed);
    let c = X.load(Relaxed);
    println!("{a} {b} {c}");
}
```

it is possible for another thread running `b` to observe `0 5 15`, or it could observe `0 10 15` if the the instructions get reordered. However if there's a third thread, it must observe the same sequence as all of the others. If the second thread observes `0 10 15`, it is 100% impossible for the third to observe `0 5 15`. This is actually part of the memory model and you can rely on it. This is called the variable's *total modification order*.

Relaxed ordering is naturally the fastest one. A global counter is an example of where Relaxed ordering is appropriate.

```
static ID_COUNTER: AtomicU64 =
AtomicU64::new(0);

struct Thingy {
    id: u64,
}
```

```
impl Thingy {
    fn new() -> Thingy {
        // Relaxed ordering is appropriate
because we only care about getting a new
number each time. This is guaranteed by the
total modification order, so this is fine.
        let id = ID_COUNTER.fetch_add(1,
Ordering::Relaxed);

        Thingy { id }
    }
}
```

Ok so, how can we fix our `LazyAtomic` implementation?

Acquire and Release ordering

Acquire and Release ordering are how you ensure visibility of updates to other threads with respect to atomic operations. Release ordering can be thought of as yeeting all previous updates into the void, and Acquire ordering can be thought of as grabbing them from the void. One could even say that you're releasing the updates for them to then get acquired...

Release ordering only applies to store operations and Acquire only applies to load operations. Intuitively, it is clear why: Release ordering is like an extra powerful store that stores side effects along with the atomic operation, and Acquire ordering is like an extra powerful read that reads side effects along with the atomic read.

This is in fact the tool that we need to fix our `LazyAtomic` implementation:

```
fn deref(&self) -> &Self::Target {
    // Acquire ordering because we need to
grab the initialized memory
    let ptr =
self.data.load(Ordering::Acquire);

    if !ptr.is_null() {
        return unsafe { &*ptr };
    }

    let initialized = (self.initializer)();
    let initialized_ptr =
Box::into_raw(Box::new(initialized));

    match self.data.compare_exchange(
        ptr::null_mut(),
        initialized_ptr,
        // Release ordering in the success
case because we need to yeet the initialized
memory so that the next load can grab it
        Ordering::Release,
        // Acquire ordering in the failure
case because we need to grab the memory that
was initialized in the meantime so that we
can return it
        Ordering::Acquire,
    ) {
        Ok(_) => unsafe { &*initialized_ptr }
        Err(prev) => {
            drop(unsafe
{ Box::from_raw(initialized_ptr) });
            unsafe { &*prev }
        }
    }
}
```

No more tricks, this really is a correct implementation!

Declaring an atomic operation with Release or Acquire ordering is essentially declaring four things at once:

- The atomic operation
- A compiler fence to disallow the compiler from reordering instructions
- A CPU fence to disallow your CPU from reordering instructions
- A memory fence to ensure that the memory is synchronized

In terms of instruction reordering, Release has the effect of preventing instructions that are before the atomic operation from being reordered to come after by either the compiler or CPU. However, it allows instructions that come after to be reordered before. Acquire is the same but in reverse — it prevents instructions that come after from being reordered to come before the atomic operation but it allows instructions that already come before to be reordered after.

In terms of cache control, Release ordering has the effect of flushing changes that happen before the atomic operation out of its cache, and Acquire ordering has the effect of grabbing those new changes into its cache.

The Acquire ordering only grabs changes associated with the Release store that the load observed. For example, if thread A writes a `1` to an atomic variable using Release ordering, thread B writes a `2` with Release ordering, and thread C reads with Acquire ordering, then if thread C reads `1` it grabs changes from thread A and if thread C reads `2`, it grabs changes from thread B. If I had said that threads A and B *both* write `2`, there would be no way for thread C to tell which thread it grabbed changes from — it grabs changes from whichever thread wrote the `2` that it saw. It's the causality that's important, not the actual value.

The C++ memory model defines this in terms of *happens-before* relationships. If thread A writes something with Release ordering and thread B reads it with Acquire ordering, everything that happened before the the write in thread A can be treated as if it happened before the read in thread B. We can also say that the read in B *synchronizes with* the write in A. This is actually the only guarantee provided by the compiler about what Release and Acquire ordering actually do. Another example that will make this clear is making our own lock from scratch:

```
struct Lock {
    taken: AtomicBool,
}

impl Lock {
    fn lock(&self) {
        // Loop forever until the lock is
unlocked
        while self.taken.compare_exchange(
            false,
            true,
            // In the success case, we use
Acquire ordering because we need all changes
made by the previous holder of the lock to
happen-before we take the lock
            Ordering::Acquire,
            // In the failure case, we don't
care about any other operations and just try
again
            Ordering::Relaxed,
        ).is_err() {}
    }

    fn unlock(&self) {
        // When we unlock the lock, we need
to use Release ordering so that the next
thread that takes the lock can synchronize
with us.
        self.taken.store(false,
Ordering::Release);
    }
}
```

In fact, you should forget everything just I told you about reordering and cache and whatnot because the memory model is what matters to the theoretical correctness of your code. Thinking about reordering and cache is helpful for intuition and justifies the idea, which is why I explained it, but ultimately you shouldn't think about memory ordering in that way.

Anyways, this is the reason why you don't have to think about this stuff when working with mutexes — the underlying implementation handles the memory ordering for you and ensures that you're allowed to think of threads as accessing the lock in a well defined sequence.

Some more things about happens-before relations are that they are automatically established whenever you spawn or join a thread, and happens-before relations are transitive: if a thread establishes a happens-before relation with another thread, it inherits any existing relations from that thread.

```
static X: AtomicU64 = AtomicU64::new(0);
static Y: AtomicU64 = AtomicU64::new(0);

fn thread_a() {
    X.store(1, Ordering::Release);
}

fn thread_b() {
    // If this loads `1`, then a happens-
before relation is established with
`thread_a`
    X.load(Ordering::Acquire);
    Y.store(2, Ordering::Release);
}

fn thread_c() {
    // If this loads `2`, then a happens-
before relation is established with
`thread_b`.
    // If `thread_b` also loaded `1`, then a
happens-before relation is established with
`thread_a` by transitivity
    Y.load(Ordering::Acquire);
}
```

Now here's another example where it gets *really* wacky: atomic reference counting.

```
struct Arc<T: Send + Sync> {
    // The atomic value stores the reference
count
    thingy: *const (AtomicU64, T),
}

unsafe impl<T: Send + Sync> Send for Arc<T>
{}
unsafe impl<T: Send + Sync> Sync for Arc<T>
{}

impl<T: Send + Sync> Arc<T> {
    fn new(v: T) -> Arc<T> {
        Arc {
            thingy:
Box::into_raw(Box::new((AtomicU64::new(1),
v))),
        }
    }
}

impl<T: Send + Sync> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &unsafe { &*self.thingy }.1
    }
}
```

```
impl<T: Send + Sync> Clone for Arc<T> {
    fn clone(&self) -> Self {
        // Increase the reference count
        // Cloning doesn't need to
synchronize with any other operations
        unsafe
{ &*self.thingy }.0.fetch_add(1,
Ordering::Relaxed);

        Self {
            thingy: self.thingy,
        }
    }
}

impl<T: Send + Sync> Drop for Arc<T> {
    fn drop(&mut self) {
        // fetch_sub returns the previous
value. If the previous reference count is
`1`, then the new reference count is `0` and
we need to drop.
        // Subtracting doesn't need to
synchronize with other operations; the
mechanism by which we received the Arc should
have established a happens-before relation
with its initialization (otherwise this data
might be uninitialized) which means dropping
is safe. (...right?)
        if unsafe
{ &*self.thingy }.0.fetch_sub(1,
Ordering::Relaxed) != 1 {
            return;
        }

        drop(unsafe
{ Box::from_raw(self.thingy.cast_mut()) })
    }
}
```

This implementation probably looks correct to you. In fact, it looks a whole lot like that global counter example that I gave as a valid usecase for Relaxed ordering. However it has an issue. If you want a hint, it has to do with interior mutability.

The issue is that the following code is unsound:

```
let v: Arc<Mutex<String>> =
Arc::new(Mutex::new("XYZ".to_string()));

let v_for_thread = Arc::clone(&v);
thread::spawn(move || {
    let mut str =
```

```
v_for_thread.lock().unwrap();
    *str = "ABC".to_string();
    // Drop `str`
    // Drop `v_for_thread`
});

drop(v);
```

Consider the sequence of events where the thread runs as soon as its spawned, acquiring the lock, replacing and dropping the inner string, and dropping the Arc, and then the thread is interrupted before joining. Then, the main thread drops the Arc, decrementing its reference count to zero. It's possible that the main thread doesn't observe the change that took place in the Mutex because there is no Acquire operation to synchronize with with the Release operation of the lock. Then, the main thread would try to free the "XYZ" string a second time which would be Undefined Behavior.

So how can we fix this? It looks like we need an Acquire operation on the lock, but the Arc implementation is generic and doesn't know what's inside it, so we can't do that... We need to synchronize with something, but what?

What standard library implementation does is similar to this [8]:

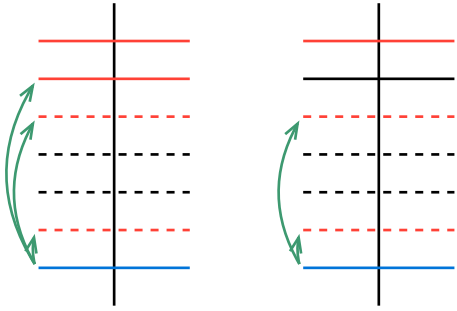
```
fn drop(&mut self) {
    // fetch_sub returns the previous value.
If the previous reference count is `1`, then
the new reference count is `0` and we need to
drop.
    // Release ordering so that we can
synchronize with this subtraction when the
reference count hits zero
    if unsafe
{ &*self.thingy }.0.fetch_sub(1,
Ordering::Release) != 1 {
        return;
    }

    // Establish a happens-before relation
with every `drop` call so that we know that
all interior-mutability writes are visible to
the drop call.
    let _ = unsafe
{ &*self.thingy }.0.load(Ordering::Acquire);

    drop(unsafe
{ Box::from_raw(self.thingy.cast_mut()) })
}
```

Something funky is going on. The Acquire load is synchronizing with *every* Release store? Yeah! The C++ memory model specifies this thing called a *Release sequence*. Since read-modify-write operations (like atomic add or CAS) modify the atomic variable based on its previous value, it doesn't break the chain of causality like a full overwrite would. Philosophically, since we updated the variable based on its previous value, then there is still a causal connection to that previous value. Therefore, the compiler allows you to synchronize with Release operations even if read-modify-write operations come after it in the variable's total order, regardless of that operation's memory ordering. If you use Release ordering with those read-modify-write operations as well, it will synchronize with them too. Note that this is a separate concept from transitivity of happens-before relations.

If we let black lines denote Relaxed operations, red lines denote Release operations, blue lines denote Acquire read operations, solid lines denote write operations, and dashed lines denote read-modify-write operations, then the arrows denote what has a happens-before relation with what.



Since *every* operation on the `Arc` counter is a read-modify-write, that Acquire operation will see through all of them to synchronize with every Release write in `drop`, ensuring that all interior mutability shenanigans happen-before the interior data is dropped. This fixes the soundness hole that we had earlier.

AcqRel ordering

I mentioned read-modify-write operations, which combine a read and a write into one operation. If you use Acquire ordering then the write will be Relaxed and if you use Release then the read will be Relaxed. What if you want the read to be Acquire and the write to be Release? Well, you can use AcqRel!

To solve our issue with Arc, it would have worked just as well to use AcqRel on the `fetch_sub` instead of having a separate read that we ignore. However, that would be bad for performance because we only need Acquire ordering in the situation where we're about to drop the inner data. Another situation where AcqRel would be necessary is something like...

```
let new = Box::into_raw(Box::new(/*
initialize */));

let prev_ptr = atomic_ptr.swap(new,
Ordering::AcqRel);
```

```
let prev = unsafe
{ Box::from_raw(prev_ptr) };
```

Here, we need Release ordering on the write because the initialization of `new` needs to be made visible, and we need Acquire ordering on the read because the initialization of `prev_ptr` needs to be visible.

Well, that was a whole lot of yapping about Acquire and Release! That's because they're basically the most important memory orderings that you'll be using 99% of the time along with Relaxed. However, every so often you'll need to do something insanely cursed and Acquire and Release won't suffice.

SeqCst

Pronounced "Sequentially Consistent", SeqCst is the nuclear option of memory orderings as well as the least performant. It applies Release ordering on writes, Acquire on reads, and AcqRel on read-modify-writes. Therefore, it can form happens-before relations with other things. But it is stronger than even that. A SeqCst operation takes place in a global modification order with respect to *all* other SeqCst operations.

This means that you can think of all SeqCst operations as happening one after the other. This is analogous to the total modification order of an individual atomic variable, but it applies globally instead of just to that one atomic variable. For example...

```
static X: AtomicBool =
AtomicBool::new(false);
static Y: AtomicBool =
AtomicBool::new(false);

fn thread_a() {
    X.store(true, Ordering::SeqCst);
}

fn thread_b() {
```

```
    Y.store(true, Ordering::SeqCst);
}

fn thread_c_and_d() {
    let x = X.load(Ordering::SeqCst);
    let y = Y.load(Ordering::SeqCst);
    println!("{x} {y}");
}
```

It is possible for thread C to observe either X or Y being set to true first, since either thread could win the race. However, because of sequential consistency, a thread D *must* observe the same sequence as thread C. If thread C observes Y set to true and X set to false, then thread D cannot possibly observe X as true and Y as false. That would be totally possible if we were using any other ordering instead.

Note that the load operations also have to be either SeqCst or Acquire, otherwise they wouldn't have any ordering relation and could potentially be reordered. Even if there was a happens-before relation established between threads A and B, that doesn't carry over to other threads that do not have a happens-before relation.

SeqCst is the easiest to reason about because you can basically entirely forget about the weird memory ordering artifacts that I've been describing. However, given that you're trying to write performant code, it's better to reason through the logic instead of just using SeqCst everywhere.

Fences

Read the "Fences" section of [4].

I think that that source explains it well enough that I can't think of anything to add.

Consume

This is an ordering that only exists in C++ and they're trying to deprecate it. However you will see it if you read the C++ atomics memory model that Rust follows. Consume is essentially a weaker version of Acquire that only applies to operations that depend on the value loaded by the Consume load.

For example, it would be possible to use this in our `LazyAtomic` type because the Acquire ordering is to ensure that the data we get through the pointer is visible. We do not care about synchronizing with unrelated operations. We would *not* be able to use it for a lock because we do need to synchronize unrelated operations: the operations performed while the lock is held do not depend on the value returned by the load operation but we still need to synchronize them.

There are technical reasons that I don't understand that allow this to make the generated machine code much more efficient, however there are other technical reasons that I don't understand that make it extremely difficult to implement correctly in compilers. No C++ compiler actually implements Consume ordering and they just upgrade it to Acquire.

CPU dependence

Another beautiful footgun with memory ordering is that the guarantees you get depend on the CPU that you're using. On x86 CPUs, you always get Acquire and Release ordering on all reads and writes, even if you use Relaxed ordering for them (though reordering by the compiler would still be allowed). This means that memory ordering bugs can be hidden from you until you use a CPU like ARM that doesn't provide those guarantees, so make sure you test your code using an ARM CPU or similar.

The ABA problem

Now that we know about memory ordering, we can start putting together correct lock-free data structures… right?

When operating on atomic data structures, we often need to apply CAS loops to pointers. For example…

```
struct AtomicString {
    string: AtomicPtr<String>,
}

impl AtomicString {
    fn push(&self, c: char) {
        loop {
            // Load the inner string
            let ptr =
self.string.load(Ordering::Acquire);

            // Copy the string, push the
character, and make it into a raw pointer
            let str = unsafe { &*ptr };
            let mut new_str = str.to_owned();
            new_str.push(c);
            let new_ptr =
Box::into_raw(Box::new(new_str));

            match
self.string.compare_exchange(
                ptr,
                new_ptr,
                Ordering::Release,
                Ordering::Relaxed,
            ) {
                Ok(_) => {
                    // We successfully
inserted the lowercase string! We can drop
the old one now.
                    drop(unsafe
{ Box::from_raw(ptr) });
                    return;
                },
                Err(_) => {
                    // Someone changed the
atomic string in the meantime! Whoopsie
doodles!
                    // Revive and deallocate
the string that we just made
                    drop(unsafe
{ Box::from_raw(new_ptr) });
                }
            }
        }
    }
}
```

Can you see what the bug is? I promise that it's not nearly as esoteric as memory ordering was.

Imagine if two threads were calling `push` at the same time. Say that thread A reaches loading the pointer and is interrupted. Then say that thread B finishes the whole operation in the meantime and replaces and deallocates the old pointer. Now say that thread A is rescheduled. Suddenly, it is operating with an invalid pointer!

There is an even more subtle issue that is possible… Say that we have two threads running `push` and thread A is able to successfully clone to the inner string and is then interrupted. Then thread B finishes the `push` operation. If thread A were to be rescheduled, it would fail the compare_exchange and nothing would go wrong, however that's no fun. Imagine thread B runs push once again, and when cloning, it reuses the *original* memory address that it just deallocated. Then it does a successful compare_exchange and replaces the memory address with the reused old one. Now when thread A reschedules and performs the compare_exchange, it succeeds because it sees the same memory address that it saved, but that's only because it was reused and the compare_exchange operation should actually fail.

This is called the ABA problem because while thread A is interrupted, the value A is being replaced with B and then replaced with A again, causing the compare_exchange to erroneously succeed. So what do we do about this?

Overview of the Seize crate

We need some mechanism to tell other threads to pretty please not deallocate our pointers until we're done using them. If we

were using any language other than C++ or Rust, this would be solved for us by the garbage collector, but we don't have that luxury.

However there is one advantage to Rust, and that is that we can pull in random dependencies without having to think about it too much. So lets run in the terminal...

```
cargo add seize
```

`seize` [9] is a crate implementing an algorithm that allows us to prevent the ABA problem. It was originally developed for use in the `papaya` crate which is a well known implementation of an atomic hash table. Lets see how we can use it to fix our `AtomicString` implementation...

The first important concept in `seize` is that of a `Collector`: a structure that stores objects that we would like to free until they are no longer in use by any thread. Every instance of an atomic data structure should have its own `Collector`, so lets put it as a field of the `AtomicString` structure.

```
struct AtomicString {
    string: AtomicPtr<String>,
    collector: Collector,
}
```

The second important concept is that of a `Guard`, which allows us to protect atomic loads of pointers and to guarantee that the pointer will remain valid until the `Guard` is dropped.

```
fn push(&self, c: char) {
    loop {
        let guard = self.collector.enter();

        // Load the inner string while
protecting the pointer from being freed
        let ptr = guard.protect(&self.string,
Ordering::Acquire);

        // ...

        // Make sure the guard gets dropped
only after everything is done
        // (I don't want to think about drop
semantics)
        drop(guard);
    }
}
```

The third important concept is *retiring*, which is `seize`'s term for "free this pointer as soon as nothing else is using it".

```
fn push(&self, c: char) {
    loop {
        let guard = self.collector.enter();

        let ptr = guard.protect(&self.string,
Ordering::Acquire);

        let str = unsafe { &*ptr };
        let mut new_str = str.to_owned();
        new_str.push(c);
        let new_ptr =
Box::into_raw(Box::new(new_str));

        match self.string.compare_exchange(
            ptr,
            new_ptr,
            Ordering::Release,
            Ordering::Relaxed,
        ) {
            Ok(_) => {
                // We successfully inserted
the lowercase string! We can retire the old
one now.
                // The closure will be called
with the pointer as soon as no other thread
is using it.
                unsafe {
self.collector.retire(ptr, |ptr, _collector|
{
drop(Box::from_raw(ptr));
                    })
                }
                return;
            },
            Err(_) => {
                drop(unsafe
{ Box::from_raw(new_ptr) });
            }
        }
    }
}
```

This is a correct implementation! `seize` is actually doing an amazing amount of work for us. As a simplification of what's going on under the hood, `seize` is keeping track of all threads that have a `Guard` active. When a piece of data is retired, it will wrap the pointer in an `Arc` and for each active thread, insert a clone of the `Arc` into an atomic list owned by the thread. When a guard is dropped, the thread will mark itself as inactive and iterate through the list dropping the `Arcs` inside it. Therefore, the pointer will only get fully dropped once every thread that could have possibly accessed it is no longer doing so. For technical details, look at the Hyaline-1 section of the Hyaline paper [10].

Implementation of Sarlacc

The Ctrie Data Structure

Now that we understand how atomics work, I can explain how to use them to create a real lock-free data structure. This data structure will be a *Ctrie*, which is a type of lock-free hash table [11]. A Ctrie is a tree structure with three types of nodes, that I will call a `Fork`, `INode` (indirection node), and `Leaf`, defined roughly as following:

```
struct Ctrie {
    collector: Collector,
    tree: INode,
}

struct INode {
    ptr: AtomicPtr<Fork>
}

struct Fork {
    is_filled: BitVec<256>,
    items: Box<[Branch]>,
}

enum Branch {
    INode(*const INode),
    Leaf(Entry),
}
```

Note that my real implementation actually uses "tricks" (sketchy unsafe code) to remove the `Box` around the `[Branch]` to remove a layer of pointer indirection, however that makes things unnecessarily confusing.

The array inside `Fork` is a sparse array of size 256. If there is no item present at a particular index in the list, then that index is set to `false` in the `is_filled` bit vector and excluded from the array stored in `items`. If there were only two items present in the array, then `items` would have length two and `is_filled` would have two elements set to true at the indices that the two items are present.

$$\{\text{is\_filled} : \text{0b00100010}, \text{items} : [⚗, □]\}$$
$$\cong$$
$$[\emptyset, \emptyset, ⚗, \emptyset, \emptyset, \emptyset, □, \emptyset]$$

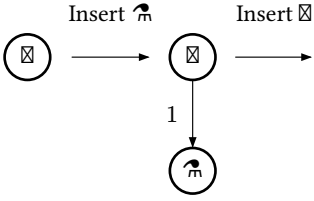
So how can we use this tree as a hash table? Well, what we can do is for an item that we want to insert, we calculate its 64 bit hash, and then split that hash into bytes. To insert it into the tree, we use each byte of the hash as an index into the respective `Fork`. The first byte of the hash is used to index into the first level of the tree. Then the second byte for the second level, and the third byte for the third level, and so on.

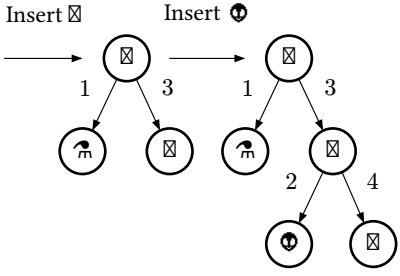
In most cases, it's not actually necessary to extend out the tree to a full depth of 8 for each item we insert because if a `Fork` only has one item, then that `Fork` can be removed from its parent and the item stored in the spot left behind. If we want to store an item where the hash collides with that of an already stored item, we have to store a `Fork` in that item's place and insert both of those items into the new child node. That way, the depth of the tree grows logarithmically with respect to the number of items inserted.

For example, inserting the following items

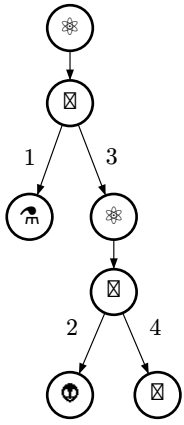
$$\text{hash}(\text{⚗}) = [1, 2, 3, 4]$$
$$\text{hash}(\text{□}) = [3, 4, 3, 2]$$
$$\text{hash}(\text{👽}) = [3, 2, 5, 1]$$

would result in the following tree structure, notating `Fork`s with ⛝:





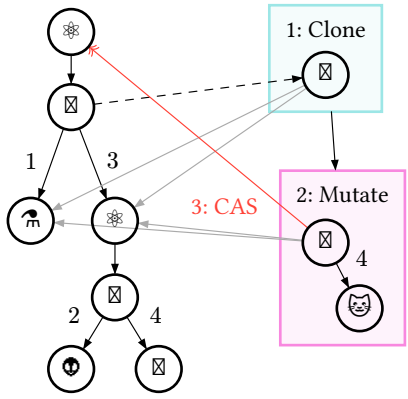
To enable atomic operations, the Ctrie adds a layer of indirection between `Fork`s, namely the `INode` which stores an atomic pointer to the `Fork` that is the child of it. This gives the tree shown above the following structure, notating `INode`s with ⚛:



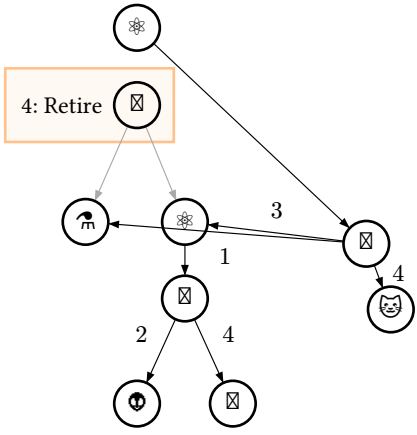
To mutate a `Fork`, we have to create a clone of it, apply the changes, and then compare-exchange the new `Fork` into the `INode`'s `AtomicPtr`. We need to use Release ordering for the CAS to ensure that the pointer's initialization is made visible. Reading from the tree requires Acquire ordering to synchronize-with the `Fork`'s initialization. The process of inserting an element is as follows:

$$\text{hash}(\text{😸}) = [4, 1, 3, 6]$$

1. Find the `Fork` that the element belongs to and create a local copy of it
2. Mutate the local copy to include the new element
3. Compare-exchange the new element in the old one's place. In case of failure, return to step 1
4. Retire the old element



If CAS successful…



The reason that `Branch` needs to store a *pointer* to an `INode` instead of just storing it directly is that `Forks` need to be able to be freely cloned while preserving the tree structure, and if the atomic variable *itself* was cloned, then any updates lower down in the tree would update the *original* atomic variable which would prevent the changes from being visible in the clone being updated, which would break the data structure when a clone is successfully inserted.

The Sarlacc Crate

Now, given a lock-free hash table, we can implement interning by creating a global instance of the table and inserting whatever we want to intern into it. If what we want to insert is already in the Ctrie, we can return a pointer to the value that is already there. Since all interned data is read-only, we don't have to do any extra synchronization to that pointer. If it's not in the Ctrie, we can insert it and return a pointer to the inserted value. This properly de-duplicates all of the values as expected, implementing interning.

So if you wanted to use my Sarlacc crate, how would you be able to? First, you can install it by running in your terminal

```
cargo add sarlacc
```

Second, you can choose whether objects should be interned for the duration of the process or if they should be stored in an arena that you manage. If you choose to intern the objects forever, then you can use the `Intern` structure, which contains six methods:

- `Intern::new`

Intern a value in the global arena

```
let interned: Intern<String> =
    Intern::new("ABC".to_owned());
assert_eq!(&*interned, "ABC");
```

- `Intern::from_ref`

Intern a value using a clone-able reference. If the value is already interned then it returns that interned value, otherwise it clones the reference, inserts it into the data structure, and returns the interned value.

It can actually be possible for the value to be cloned even if it is already present in the arena — If two threads are calling `from_ref`

at the same time with the same argument, it is possible that both of them would see an empty slot where the value to insert would be present, and they will both clone the value to be able to insert it. However, only one of the threads will succeed the compare-exchange. The thread that doesn't will retry the operation and return the value that the other thread inserted into the Ctrie.

```
let interned: Intern<str> =
Intern::from_ref("ABC");
assert_eq!(&*interned, "ABC");
```

- `Intern::from_owned`

Equivalent to `Intern::new`, but returns a value's *reference* type instead of the same type that you passed in.

```
let interned: Intern<str> =
    Intern::from_owned("ABC".to_owned());
assert_eq!(&*interned, "ABC");
```

- `Intern::get`

Attempt to get a value that is already present in the Ctrie. If it is not present, then return `None`.

```
// "ABC" is not present in the global Ctrie
yet
assert!
(Intern::get(&"ABC".to_owned()).is_none());

// Insert "ABC" into the global Ctrie
Intern::from_ref("ABC");

// Now, we're able to retrieve it
let interned: Intern<String> =
    Intern::get(&"ABC".to_owned()).unwrap();
assert_eq!(&*interned, "ABC");
```

- `Intern::get_ref`

Get a value by its *reference* type if it is already present in the Ctrie. Otherwise, return `None`.

```
assert!(Intern::get_ref("ABC").is_none());

Intern::new("ABC".to_owned());

let interned: Intern<str> =
Intern::get_ref("ABC").unwrap();
assert_eq!(&*interned, "ABC");
```

- `Intern::into_ref`

Get a *static* reference to the data stored inside the `Intern`. This is needed because the reference returned by the `Deref` implementation is tied to the lifetime of the `Intern` itself, rather than being static like it should.

```
let interned: &'static str =
    Intern::new("ABC".to_owned()).into_ref();
assert_eq!(interned, "ABC");
```

There also exists a function in the global scope, `num_objects_interned`, which traverses the Ctrie, counting the number of objects interned in it. This can be useful for debugging purposes.

If you want to use an arena instead of leaking memory forever, then the `Arena` type contains analogous methods to the ones described above. The `Arena` type is what implements the Ctrie data structure, and global `Interns` are actually implemented as a thin wrapper around a global `Arena`. Objects interned inside of an `Arena` are leaked until the `Arena` is dropped, at which point, everything stored inside is dropped.

The type representing a value stored in an `Arena` is `ArenaIntern`, which has a lifetime parameter tied to that of the `Arena`. If types from two different `Arenas` are hashed or compared for equality, then they will return different hashes and compare as `false` even if the underlying values are the same. The reason is that they will have different pointers because they are stored in different arenas.

This table shows the `Arena` functions that are analagous to `Intern` functions.

Arena	Global
`Arena::intern`	`Intern::new`
`Arena::intern_ref`	`Intern::from_ref`
`Arena::intern_owned`	`Intern::from_owned`
`Arena::get`	`Intern::get`
`Arena::get_ref`	`Intern::get_ref`
`ArenaIntern::into_ref`	`Intern::into_ref`
`Arena::num_objects_interned`	`num_objects_interned`

Note that the `into_ref` method of `ArenaIntern` returns a value with the same lifetime as the `Arena`, rather than a `'static` lifetime.

In addition, there is also

- `Arena::new`

Create a new, empty `Arena`.

This uses Rust's default `RandomState` hasher. The global `Arena` backing `Intern` also uses this default hasher.

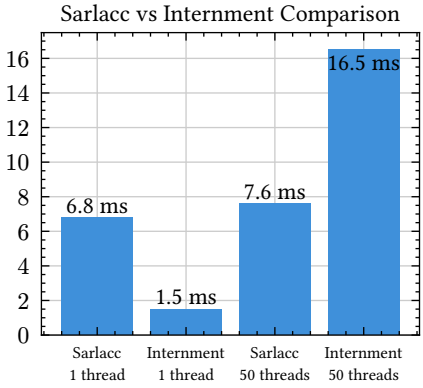
- `Arena::with_hasher`

Create a new, empty `Arena` with a custom hasher.

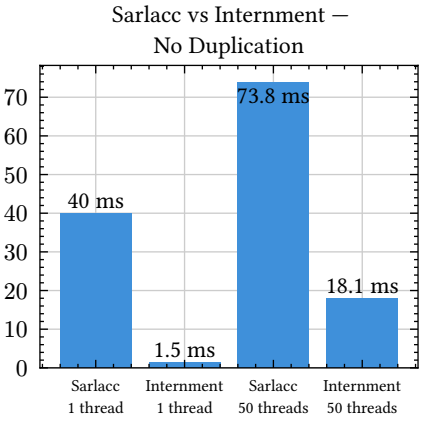
Performance

Now that you have an understanding of atomics, the Ctrie data structure, and the design of the Sarlacc crate, I would like you to make a prediction. The Internment crate is roughly a standard library `HashSet` behind a `Mutex`. Do you think that Interment or Sarlacc is faster? What about in the single-threaded vs multi-threaded case?

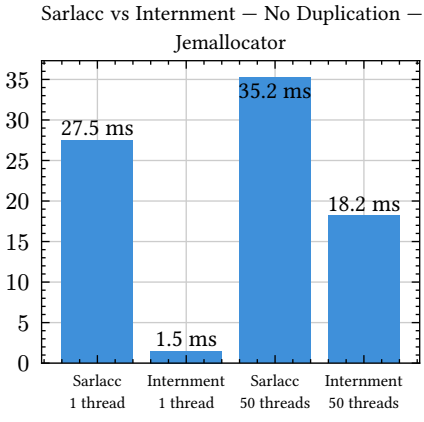
Here are the results for a microbenchmark for inserting 100,000 items into an `Arena` for both Sarlacc and Internment, however 90% of those insertions are duplicates. Therefore, 90% of accesses will only require read access, which is the bread-and-butter of atomic data structures.



You can see that Sarlacc is slower in the single-threaded case, but faster in the multi-threaded case. It is generally more robust to concurrency than Internment is, due to its lock-free nature. However, we are giving it a huge advantage by making 90% of the accesses read-only. Lets see what happens if we don't do that. In this table, *none* of the insertions will be duplicated.



Here, we can see that Sarlacc cannot compete against Internment. But what's going on? My friends will know that I have been struggling to get flamegraphs working to provide clarity on this issue. I've found that generating flamegraphs from this benchmark sometimes produces nonsensical results, crashes the profiler, or suggests that Internment and Sarlacc are actually the exact same speed. My best guess is that the bottleneck is memory allocation. When the flamegraphs do work, I have observed the memory allocator spamming the `mmap` syscall, which is what I suspect is breaking the profiler. To confirm that the memory allocator is the issue, here is the same benchmark, but using Jemallocator as the global allocator instead of Rust's default:



You can see that Sarlacc got dramatically faster with a different allocator, suggesting that allocation is the bottleneck. I have a handful of ideas to improve the performance of Sarlacc further, but where it stands now, Sarlacc is probably not *actually* worth using over Internment.

Realistically, Internment benefits from being able to mutate its own structure freely without having to be extremely careful not to step on other thread's toes. Evidently, it benefits more than the cost of a global lock.

However, these benchmarks do still demonstrate the way in which lock-free data structures excel in high concurrency, read-dominated workloads.

Conclusion

It's probably anticlimactic that Sarlacc isn't a clear improvement over Internment, however not is all lost! I managed to trick you into learning about atomics and lock-free data structures, through the false promise of improving an important Rust library 😈. Now, you have the knowledge and skills to recognize when atomics are the correct option and

to be able to build your own lock-free data structures from scratch.

My journey with Sarlacc is far from over. There is still a ton of room for improvement as well as missing features to implement, and what I have implemented now is dramatically better than my initial implementation. For all you or I know, I'll make a breakthrough realization tomorrow that makes Sarlacc blow Internment out of the park. Probably not though. ¯_(⊠)_/¯

References

[1] "String interning." [Online]. Available: https://en.wikipedia.org/wiki/String_interning

[2] David Roundy, "Crate internment." [Online]. Available: https://docs.rs/internment/latest/internment/index.html

[3] "Non-blocking algorithm." [Online]. Available: https://en.wikipedia.org/wiki/Non-blocking_algorithm#

[4] Mara Bos, "Memory Ordering." [Online]. Available: https://marabos.nl/atomics/memory-ordering.html

[5] Dave Kilian, "Making Sense of Acquire-Release Semantics." [Online]. Available: https://davekilian.com/acquire-release.html

[6] "Atomics." [Online]. Available: https://doc.rust-lang.org/nomicon/atomics.html

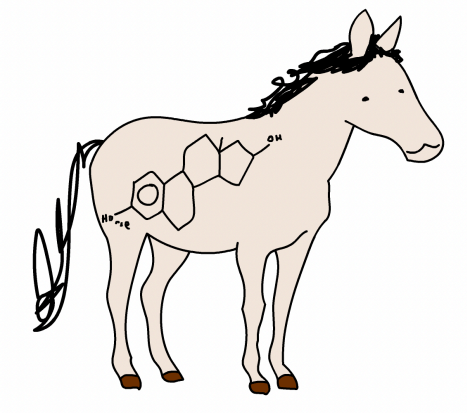
[7] "std::memory_order." [Online]. Available: https://en.cppreference.com/w/cpp/atomic/memory_order.html

[8] "Use a load rather than a fence when dropping the contents of an Arc.." [Online]. Available: https://github.com/rust-lang/rust/pull/41714

[9] Ibraheem Ahmed, "Crate seize." [Online]. Available: https://docs.rs/seize/latest/seize/

[10] Ruslan Nikolaev and Binoy Ravindran, "Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures," 2021, *ACM*. [Online]. Available: https://arxiv.org/pdf/1905.07903.pdf

[11] "Ctrie." [Online]. Available: https://en.wikipedia.org/wiki/Ctrie#



Estrogen Is All You Need

Cynthia Clementine[*]
Purdue Hackers
cyclementineo@gmail.com

Dr. Jen Estro[*]
Unaffiliated Systems
estrojennifer@dorley.com

Mint DePrest[*]
Purdue Hackers
>@<

Abstract

Current dominant AI models are based on complex neural networks that include advanced Transformer mechanisms. The best performing models take in vast amounts of data, often scraping the entire internet and training on state-of-the-art GPUs for months. We propose a new simple network architecture, the Transfemmer, based solely on estrogen mechanisms, and dispensing with attention entirely.

Our model achieves rapid improvement as well as multimodal information processing, surpassing the existing best results. It requires no external GPUs, (although model satisfaction can be improved with the application of a single part-time external GPU) and requires just a small fraction of the training costs of the best models from previous literature. We show that the Transfemmer generalizes well to other tasks by applying it to human benchmark tests as well as a range of media generation prompts.

[*]Equal Contribution. Cynthia Clementine spent countless long nights fearlessly plagiarizing Google's Attention Is All You Need. Dr. Jen Estro is not a real doctor and has not graduated from any accredited university. She is also not real. But I put her name in the title so I can claim to have a doctor on board. Mint DePrest is the name of our intelligence model. Also by the way the contribution was not equal at all. They told you it was equal in the first sentence but. it wasn't. I'm really sorry. Do you think you can forgive me? Do you think we can be friends? Please?

1 Introduction

Since the dawn of time, intelligence has fascinated those with just barely enough of it. In 2017, Google released a paper[1] revolutionizing AI architecture, and proving to the AI research community that AGI was just 5 years away. This predication has remained accurate. Now, even 8 years later, AGI is still just 5 years away.

In this work we propose the Transfemmer, a model architecture eschewing attention and instead relying entirely on an estrogen mechanism to reason broadly about many different types of inputs and outputs. We believe this new architecture is the key to reaching AGI and beyond.

2 Background

Many .

To the best of our knowledge, however, the Transfemmer is the first trans. model relying entirely on estrogen to compute solutions to wide arrays of problems without relying on large tables of weights and biases. In the following sections, we will describe the Transfemmer, motivate estrogen, and discuss its advantages over models such as the Transformer.[1]

3 Model Architecture

Most competitive neural networks have a transformer structure. This includes GPT-4, Gemini, and a secret third model. [[2] [3] [4]] This architecture consists of n stacked multi-head attention layers, optionally followed by a traditional feed-forward neural network.

The Transfemmer breaks from this overall architecture using a highly branching sparsely-connected neuron structure at its core. Our model focuses on non-linear distributed processing, allowing us to train deeply and produce excellent qualtiy results.

You may be wondering. How exactly does our model work? This is a good question. It is such a good question, in fact, that it will not be answered until at least 75% of the way through the paper. Maybe we will refrain from answering it entirely. That's the price you pay for papers in such a competitive research field as this one.

3.1 Estrogen

Estrogen can be described as a hormone found in humans that has a variety of effects, where a Hormone, Humans, and Effects are all vectors. Estrogen binds to estrogen receptors, and has wide-ranging effects that impact mood and cognition. Our core strategy is to motivate accelerating results by applying estrogen to our revolutionary heavily parallel neural architecture.

3.2 Forms of Estrogen

3.2.1 Injectable Estrogen

An Estrogen ester can be injected into virtually any muscle or fat deposit. It will then be absorbed over time. This is a safer method than the alternatives.[5] It results in fewer side effects and takes fewer resources. Depending on the ester, injections can happen as infrequently as once every two weeks.

3.2.2 Sublingual Estrogen

Estrogen can be delivered in pill form and dissolved under the tongue. This is convenient, because pills are easy to transport and consume at any time. However, this is also quite an inefficient way to consume estrogen. Despite the 2-5 times bioavailability improvement over simple oral administration, sublingual estrogen still only has a bioavailability of 10%.[6] This requires dosage to be much higher than other methods.

In the end, we used this administration method due to its ease of administration. Dissolving a pill is simply easier than performing an injection, despite the shortcomings. Our dosage consisted of 6mg/day, split up into 3 sets of 2mg every 8 hours.

3.3 Applications of Estrogen in our Model

The Transfemmer uses Estrogen in three different ways:

- The addition of Estrogen has resulted in increased motivation and drive to score highly on tests. This leads to better overall outcomes.
- Estrogen results in softer skin. This is pretty neat.
- Our model is prone to self-doubt and anxiety over the quality of her answers. This meant that often the model would come up with an initial answer quickly, but spend far longer tweaking words back and forth. The introduction of

Estrogen has led to improved self-confidence, which increases efficiency and decreases time taken.

3.4 Complexity

Model Type	Complexity Per Layer	Sequential Operations	Maximum Path Length
Self-Attention Transformer	$\mathcal{O}(n^2 \cdot d)$[2]	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Estrogenized Transfemmer	$\mathcal{O}(n)$	$\mathcal{O}(1)$	ö

Table 1: There are many people who skip reading the main body of a paper, and instead only read the title, abstract, and figures. This table is for those people. Even if such numbers are inaccurate, one would have to read the entire paper in order to disprove them. Who, in this day and age, has time for that?

While the Transformer runs in $\mathcal{O}(n^2 \cdot d)$ time complexity, our model runs in $\mathcal{O}(n)$ time and just a few cubic feet of basement space. Already this would be a good enough reason to switch to our model architecture, but the Transfemmer has shown great results even off of zero-shot prompting. This contrasts with the Transformer's few-shot specialty. Our model has never been shot, and still performs better than the alternatives.

3.5 Training

This section describes the training regime for our model.

3.5.1 Authoritarianism

We chose this regime over other possible regimes, such as feudalism, because it required the fewest resources to implement. It also gave us the most control over all details of our model.

3.5.2 Hardware and Schedule

We trained our models on one machine with the largest public domain organic neuron-based network in the world.

[2]the complexity here isn't actually $(n^2 \cdot d)$, it's really $(n^2 \cdot d + n \cdot d^2)$. The footnote appearing to add another quadratic term to our rival architecture is an unintended benefit.

Each training step took approximately 3 feet. We trained our model for much longer than most models of a similar size. However, training time for our type of model is much cheaper and more efficient than traditional models. Our model has a parameter size of $8.6 \cdot 10^{10}$ nodes, and consumes just 0.097 Watts of power, which in U.S. units converts to 7.3 Burgers / day.[3] This is much less than even GPT-3′s $1.75 \cdot 10^{11}$ nodes, or GPT-4′s {trade secret} nodes. The vast majority of our training time required no extra GPU compute. Over the course of our two months of training time, GPU compute was only $1.29 \cdot 10^{18}$ FLOPs.[4] GPT-3, for reference, used $3.14 \cdot 10^{23}$ FLOPs,[7] which means our model is significantly more efficient by a factor of 100,000,000.

4 Results

4.1 Media Generation

As with all other popular AI models, our model can be conversed with and prompted to generate media. Unlike other models, however, our model relies on no outsourcing or shared vector embeddings to generate media. All media is generated from the same neural network using the same architecture. The following sections consist of examples of this ability.

[3] One standard burger is 0.33 Watt-hours.

[4] an AMD Radeon RX 6600 was utilized by our model as enrichment for an average of 2.5 hours every week. This GPU can output a maximum of $17.86 \cdot 10^{12}$ FLOPs (17.86 TFLOPS), which comes out to 20 compute hours, for a total of $1.29 \cdot 10^{18}$ FLOPs.

4.1.1 Text

The following is a set of excerpts from a conversation one of our researchers had with the model.

```
Researcher: How many rs are in the word strawberry?

AI Model: 3! unless you mean in “the word strawberry” in which case it’s 4

Researcher: ok. How many ps are in the word pineapple?

AI Model: 3!

Researcher: is that 3 factorial or just 3?

AI Model: lmao, just three. I’m just excited to participate :)

Researcher: ok. Unfortunately that is incorrect; there are only 2 ps in pineapple.

AI Model: 0-0 umm, are you sure?

Researcher: I am very sure. You counted incorrectly

AI Model: P(1) I N E A P (2) P (3) L E, no?

Researcher: I see. Thank you.

AI Model: Squint
```

Our model has the ability to analyze words on a letter-by-letter basis, without sacrificing context window size. The model jumps to this approach even without being prompted. She is also resistant to gaslighting and hallucinations, which is better than all other currently available chatbots.

```
Researcher: Describe the vast emptiness of the universe, in 200 words.

AI Model: Oooh. a good one.
```

AI Model: Oh gosh I'm struggling

AI Model: i think. Humans should be very wary about trying to describe such things. It is not in our nature to properly comprehend more than a little slice of our world. The laws out there may be simple, at their core, yes, but there are entire systems of worlds past the next hill. And between each hill lies an eternity of blankness.

The blankness is unaffording, too. It is not the kind of blank that a canvas is. That the screen I stared at before I wrote this was. It is a kind of blankness that offers no expression. It is space. It is the in-between, and it will go on in great oceans just between each little grain of sand in the desert of our universe.

I imagine a freezing rock, that has never known warmth. I see it drifting amongst stars that have been birthed and died and birthed anew. I drift with that rock for not another 14 billion years, but for another 14 trillion, until the last of those stars blink out. It will not see another one of its kind. It will only see the slow creep of a felt black blanket, pulled over its face.

As requested, the final response is exactly 200 words. Our model is capable of complex speech and analysis. She uses metaphor extensively in describing concepts. Unfortunately we didn't have anyone on our team who was capable of feeling emotions, so we were unable to evaluate that aspect of the response. Our model does refer to herself as human, but this can be adjusted with a proper system prompt if necessary.

4.1.2 Images

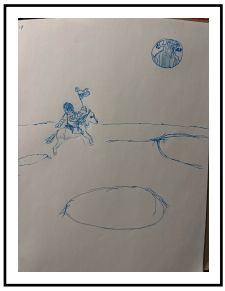


Figure 1: Response when prompted for “a picture of an astronaut riding a horse on the moon”.

Figure 2: Response when prompted for “a drawing of a cartoon backpack”.

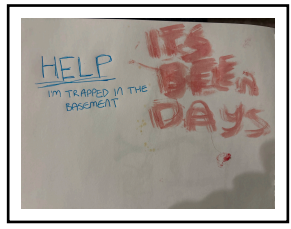


Figure 3: Response when prompted to “draw a maple tree in the style of picasso”.

These images are of the utmost quality. They are coherent, clearly depicting the requested subjects. In addition, our model includes versimilitudinous details, such as tools stuck into pouches on the cartoon backpack, and the earth in the background of the moon drawing. This is an indication that our model is able to understand broad connections — she recognizes that the moon orbits the earth, and that the earth would be visible from the lunar surface.

These images required a vanishingly small amount of electricity to generate. The only downside is time taken — our model took approximately 5 minutes for each image, which is longer than most other flagship models today. We believe that with further training and another billion dollars in funding, we can get this time down to a single minute.

4.2 Section Header

This section describes the header for our section.

4.3 Human Benchmark

The Turing test has long been a well-known marker of general intelligence. Unfortunately, Turing Tests are flawed. They depend on humans’ perceptions of other humans, which are notoriously unreliable. Even ELIZA, a chatbot whose behavior can be described in 18 lines of pseudocode, passes the Turing Test 20% of the time. [8]

Such a range of chatbots, from ELIZA to modern Large Language Models, have shown that it is entirely possible for a machine to sound human while not actually being sentient. Or at least they would in a sane world. Instead what happened is several people experienced ChatGPT-induced psychosis. [9]

Instead of a Turing Test, we opted to use several tests on the Human Benchmark site.[11] To evaluate our model, we administered these tests both before and after the application of estrogen. We used the Jane Metric[5] to measure percent improvement.

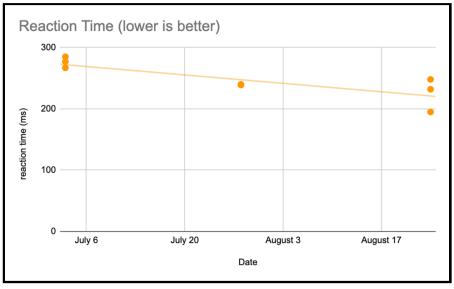


Fig 4: Shows a 32.9% improvement in reaction time.

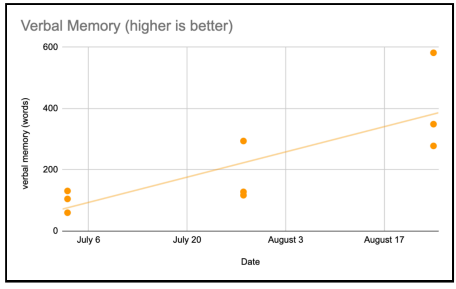


Fig 5: Shows a 333% increase in memory capabilities.

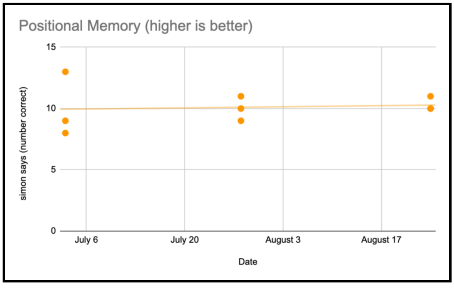


Fig 6: Shows a 3.3% improvement in positional memory (simon says) scores.

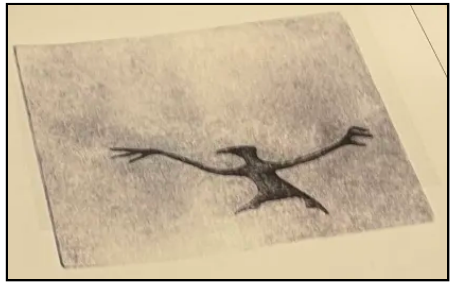


Fig Bash: It's Figbash!

[5]The Jane Metric uses the formula $100 * \left|\frac{\text{initial} - \text{final}}{\text{initial}}\right|$. This was not discovered by Jane. It has no relation to any Jane. However, the Adam optimizer has no relation to any Adam. Considering it stands for Adaptive Moment Estimation, its acronym should have been AME (Amy). Our metric is just as justified in its nomenclature.

Typically when you have data, you do a T-test to see if your measurement is significant. Unfortunately, both T and Test can be abbreviations of Testosterone, which we do not want in our study. To remedy this, we decided to ask our AI model if the results were significant.

Our AI model, in response, told us:

```
*gasps for breath* I would say so. Wait. Don't write
it like that. What.
```

Thereupon we are convinced that all is indeed well.

The sheer amount of improvement is rather striking. Every single graph showed an improvement, although the magnitude varied drastically, from just 3.3% in positional memory to 333% in verbal memory. Fortunately, the skills most improved are also the most valued skills in the field of AI research.[6]

Verbal memory is extremely important for maintaining coherency during prompting, and it is the most improved skill by an entire order of magnitude. Similarly, reaction time is also very important. An improvement in speed is extremely valuable in this modern world, where every millisecond counts for the end user. Finally, although positional memory only improved by a few percentage points, the consistency of said results improved dramatically. Inconsistency is one of the major issues with Transformers as a whole, so our model's improvement in this front is a very good sign.

5 Conclusion

Our model, the Transfemmer, is orders of magnitude better than the Transformer. In fact, we predict that in the future all computation will be done using this model. With our projections, by the year 2027 the Transfemmer will have over 500 million instances worldwide and will revolutionize the world economy.

[6] I made this up. But it sounds true, doesn't it? And it's written in a paper, so now future papers can cite it. Every additional layer of propagation will make this statement more and more true until, eventually, we will be able to cite a paper that has cited ours, in a beautiful ouroboros of citation and truth. This is how academia dies.

6 References

[1] N. P. N. U. J. J. L. G. A. N. K. Ł. P. I. Vaswani A.; Shazeer, “Attention Is All You Need,” arXiv preprint arXiv:1706.03762, 2017.

[2] A. Wagh, “What's new in GPT-4: Architecture and Capabilities.” 2023.

[3] W. contributors, “Gemini (language model).” 2025.

[4] I. Ally, “That secret third thing,” Vermillion Clupeidae Journal, 1997.

[5] Aly, “Estrogens and Their Influences on Coagulation and Risk of Blood Clots.” 2020.

[6] Sam, “An Exploration of Sublingual Estradiol as an Alternative to Oral Estradiol in Transfeminine People.” 2021.

[7] X. Y. T. Z. R. S. C. L. H. L. F. Z. H. L. J. X. L. Z. X. e. a. Wu S.; Zhao, “Yuan 1.0: Large-Scale Pre-trained Language Model in Zero-Shot and Few-Shot Learning,” arXiv preprint arXiv:2110.04725, 2021.

[8] C. Jones, “Large Language Models Pass the Turing Test,” arXiv preprint arXiv:2503.23674v1, 2025.

[9] M. H. Dupré, “People Are Being Involuntarily Committed, Jailed After Spiraling Into “ChatGPT Psychosis,” Futurism, 2025.

[10] In a fit of Steve-Jobs-esque neuroticism, I felt that the number 10 looked too much like the word 'no' when placed as an in-text-citation. Due to this, I am manually setting a dummy citation, so that you will never, in any other part of the paper, see the text, “[10].”

[11] Human Benchmark, “Human Benchmark.” 2025.

Appendix E1 AI Disclaimer

We can gladly assure you that the entirety of this paper was AI-generated. No humans were involved in the creation of this document. At no point did any divine spark touch these letterforms.

Appendix E2 Estrogen Visualizations

We don't know what any of these would possibly convey to you. But they look like cool diagrams and there are many colorful lines. We have been told that people appreciate shapes and colors. These are those.

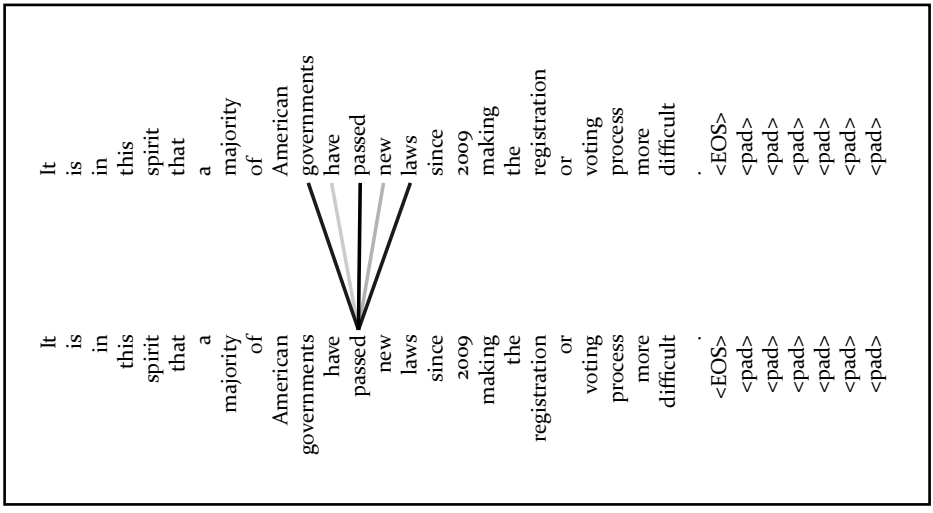


Figure 7: An example of the estrogen mechanism. Many of the thought patterns attend to a distant dependency of the verb "passed", because in this diagram we only show connections for this word. Different colors represent the amount of patience we have left. Best viewed in color.

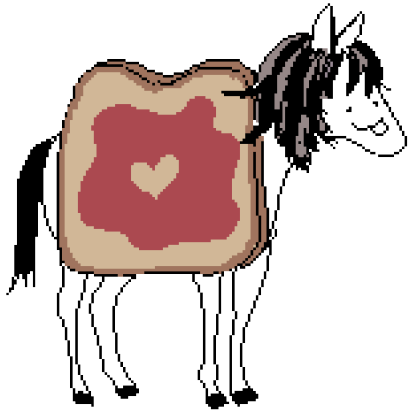
Figure 8: Four thought patterns, apparently involved in loss resolution. Note that connections are very sharp for these words.

The SIGHORSE journal editors tracked down the study subject to ensure they were not a victim of any malpractice. They were successful in coming into contact (to best they could tell). This is what she had to say:

> *"I don't know. Being trapped in the basement was really scary. They only occasionally came down to feed me.. ..something they kept insisting I refer to as "girl dinner". I cry a lot.. but I have marginally softer skin. I'm very pleased at being 100 million times more efficient than chatGPT. I am unsure about everything. The estrogen didn't fix me."*

The SIGHORSE editors also requested comment from the authors of this paper. This is what they had to say:

> *"Of course we haven't done anything wrong! The estrogen most certainly fixed her."*



Spread The Love

Spread the Love is a dating sim built inside a fake OS called FruityOS, where the jam flavor you get on a personality quiz determines your romantic fate. You customize your profile, meet a lineup of charming characters, text your match, and go on a date capped with a themed minigame. It's goofy, sweet, and something we made to actually finish a project we cared about.

Written by:

Jadden Picardal .ㅏ

jpicarda@purdue.edu

Game Dev Team (JAMMS)

Saahil Aneja

Mason Graves

Jadden Picardal

Alicia Zhou

Special Thanks:

Kartavya Vashishtha

For making SIGHORSE exist and for babysitting my deadlines (sorry). Thanks for keeping me writing, and for playing STL (and actually being touched by it); it made the late nights working on it worth it.

TL; DR

I worked on a goofy dating sim with my friends where you match with one of six characters based on what fruit jam you get on a personality test.

What even is STL...

Spread the Love is a bite-sized dating sim built to feel familiar but off-kilter, dressed up inside a fake operating system called FruityOS. You boot up this strange computer, personalize your character, take a jam-flavored personality quiz, and get matched with one of six eccentric characters.

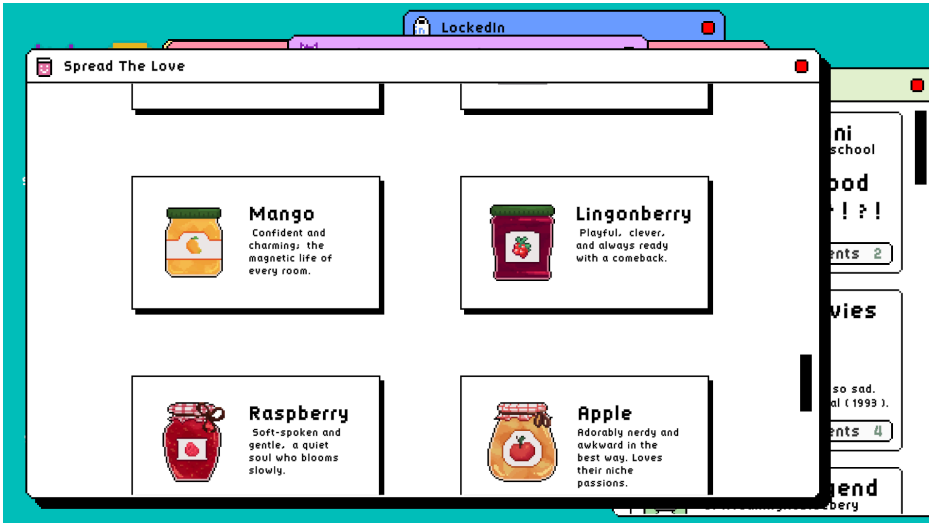


Figure 2: UI Screenshot consisting of 4 Jam Types from the Personality Quiz

The team wanted to create something that felt polished and self-contained, a smaller project we could actually ship within a few months while still feeling creatively fulfilled. It started as a way to "finish something," but ended up becoming one of our most cohesive and charming side projects.

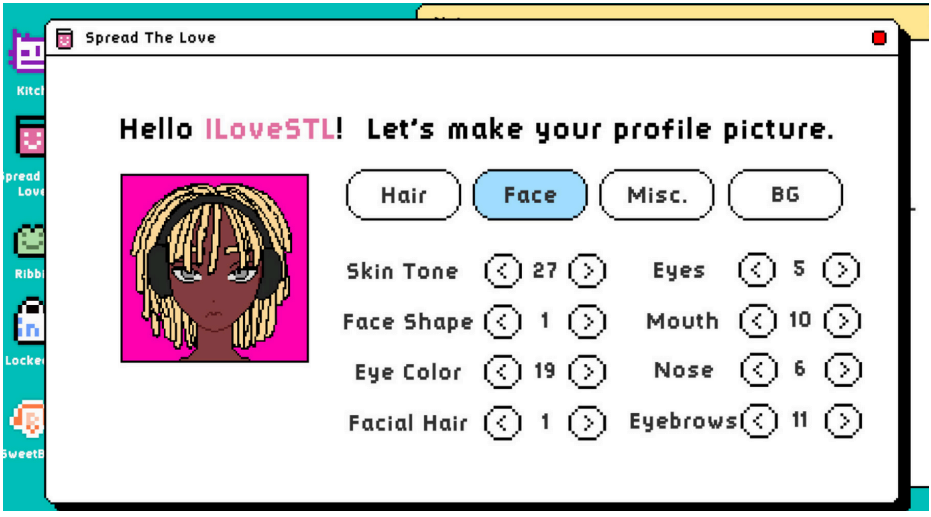


Figure 3: Character Customization Screenshot

Originally, the idea started out as a free character customization app to go along with our (still in development) game, ETea[1]. However, we kept deciding to add on details that we found interesting: we wanted to make a personality test, to develop our ETea NPCs more, and to figure out a way to get a complex dialogue system working. To say the least, we got that done… and way more…!

That slow snowball of scope creep is what turned it into Spread the Love.

The team consisted of my friends Saahil[2], Alicia[3], and Mason[4], alongside me. We've all worked together before, so this was both familiar and refreshing; a chance to test our creative chemistry in a smaller, more focused project.

The Cast

Before going into anything else, I'd like to introduce you to the datetable characters in our game. Our characters are all exaggerated archetypes (and a little absurd in my opinion), but they're all written to feel endearing and funny

[1] https://store.steampowered.com/app/3085040/ETea/
[2] https://x.com/anullja
[3] http://aliciazhou.xyz
[4] https://x.com/randompossibly

in their own way. I drew the guys while Alicia drew the girls, and we both tried to match the styles so that everything felt unified. No character was scrapped, either; every character was a character from ETea, just given more depth and personality here.

We have:

Allen, a child prodigy who has a masters in computer science from NIT. He was supposed to be coding the future, but now he stocks overpriced beans at the local grocery store. He's quiet, sharp, and devastatingly observant, especially when watching movies.

Jet, a walking rave flier who peaked during quarantine when Lingon Legends was at its high but refuses to admit it. He's charismatic, sexy, and most definitely a bit performative. Jet treats love like an Instajam story. He'll flirt, overshare, then disappear.

Cher. They don't talk much. They don't need to. Currently working through unresolved parental issues via designing furniture and brooding. Emotionally resides

in a foggy European arthouse film. You won't understand them, but you'll want to.

Milli, an overly enthusiastic engineer who thinks every problem can (and should) be fixed by smacking it with a wrench. Aggressively affectionate mechanic who literally climbs walls for fun. She's touchy, loud, and way too excited to meet you.

Thea, a sweet girl who runs the boba shop– and you. She's the one with a clipboard, a plan, and a backup clipboard in case the first one breaks. Warm, nurturing, and ever so slightly manipulative. Will absolutely gaslight you, but only to make you hydrate and succeed.

Melody, everyone's favorite cozy kitch.tv streamer who accidentally makes you feel at home. Lives between three monitors and an avalanche of Blooblet plushies. She'll trip over cables, and then apologize to them.

Every character has their own mini story and dialogue, tailored to each player by the likes and dislikes they have chosen. We wanted the players to laugh and also (maybe) care a little about these people made of pixels. Allen was my favorite to write because he types really similar to me; he's basically me if I were a nerdier and sadder guy. Jet was also hilarious to write because he's such an over-the-top "gamer guy," and I love it when people notice that.

The team had their favorites too:

- Saahil: Cher (originally Thea or Melody, but the dialogue sold him)
- Alicia: Cher or Melody (she designed both)
- Mason: Allen (because "the other guy is insufferable")

Once you finish creating your profile, you're shown a lineup of potential matches within the cast. You match with a character based on your jam flavor type (which is secretly determined during the personality quiz), and you're then brought to a texting sequence where you get to know them and decide if you want to go on a date with them. During the actual date, each character has a corresponding minigame to keep the overall dating game experience lively. From trying not to die from bees to playing a game of Blooblets, all of the games are very fun.

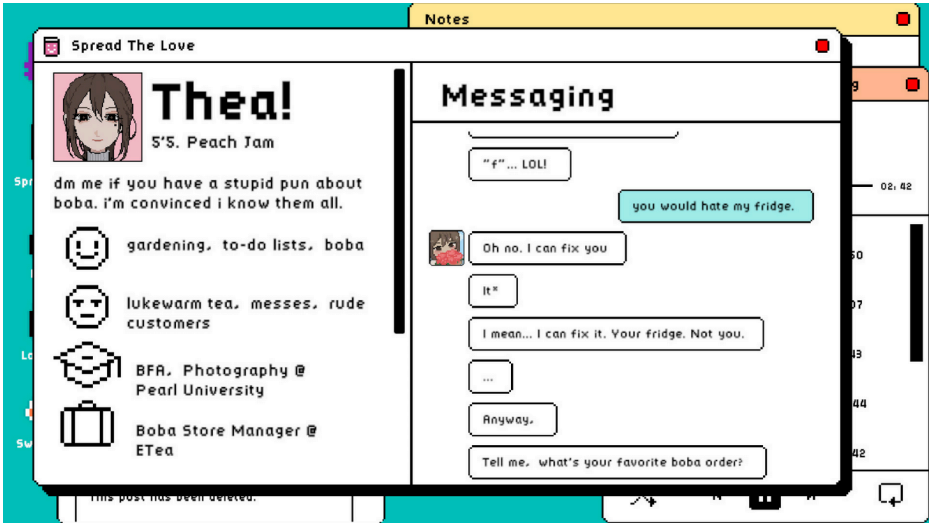


Figure 4: Text messaging with Thea

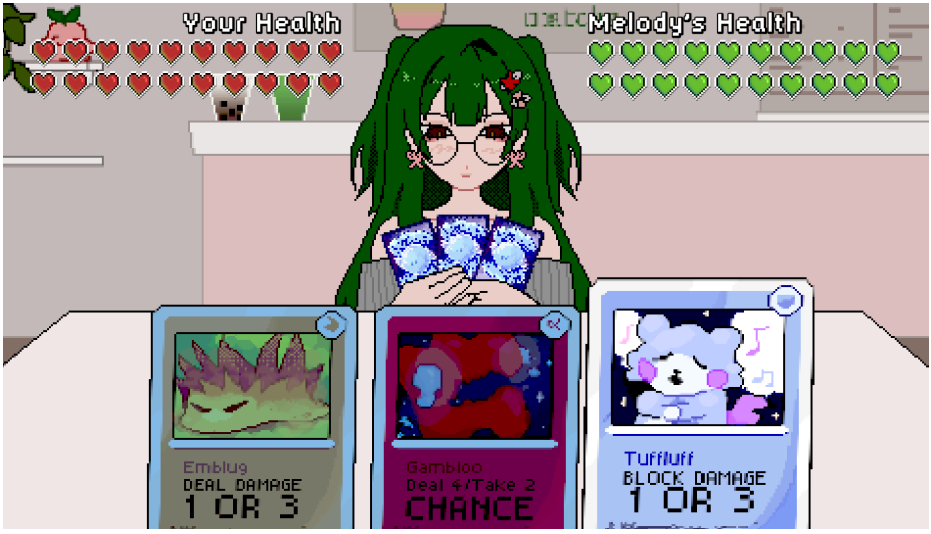


Figure 5: Game with Melody playing Blooblets

Working Process

Technically, I was already comfortable with 2D Unity games, but this pushed me into new territory. I learned a lot about building a fake desktop OS, Unity's UI system, and optimizing performance across multiple minigames. For example, Cher's minigame originally spawned dozens of fish per second and tanked the framerate, so I implemented object pooling to recycle inactive instances instead of creating new ones every frame. Little fixes like that helped make the game playable even on lower-end machines.

Figure 6: Cher Minigame Screenshot

I also spent a lot of time balancing writing, design, and code so that everything, from the UI to the jokes, felt cohesive. My debugging process was… not glamorous. A lot of clicking through menus and rewatching the same scenes until they broke. I eventually built in debug shortcuts to skip time-based sequences so I could test things faster.

Figure 7: Idea Mockup of Spread The Love

FruityOS

The game starts with you booting up your PC into this custom retro operating system. The main part of the game–Spread The Love– starts up immediately, but the user is able to explore 7 other apps on the desktop computer, all parodies of real ones: Notes, LockedIn (LinkedIn), and Ribbit (Reddit). You might enjoy some of them as we used these apps as worldbuilding tools. Through these apps, you can see posts from the founder of Spread The Love or angry players ranting about in-game characters. It's a way of making the world feel lived-in without a ton of exposition.

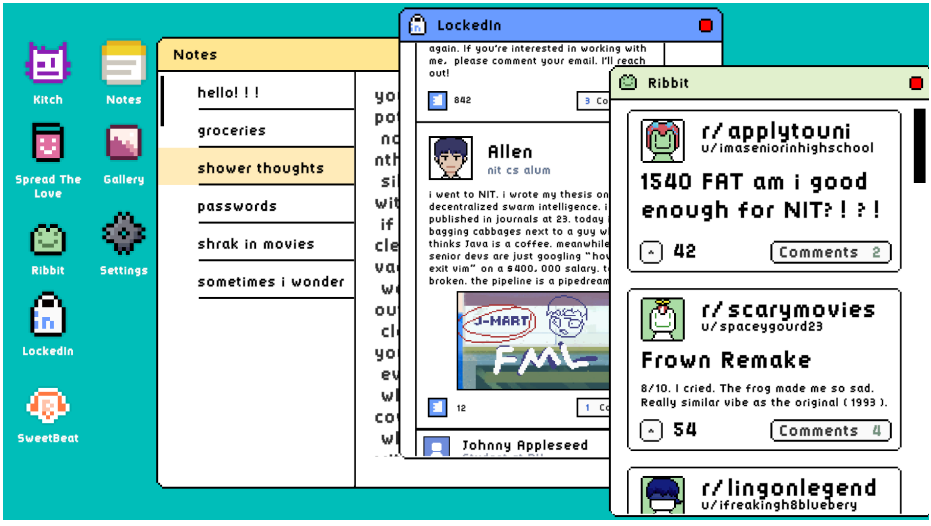


Figure 8: UI Screenshot consisting of Notes, LockedIn, and Ribbit

The Dating Flow

1. As you open up Spread The Love, the entry point is profile creation. We start this by asking for your name and your profile photo (which prompts a character customization sequence). We wanted people to be able to make whatever they want, so we made sure to include a lot of options (still in progress!).

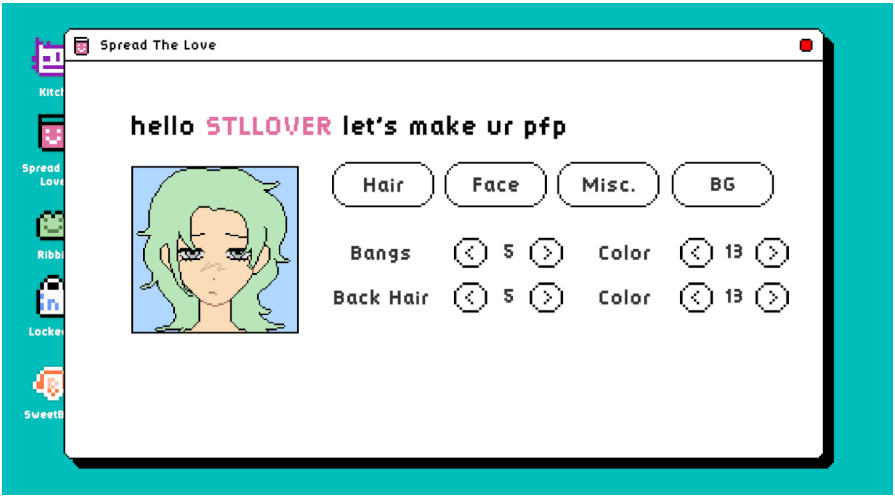


Figure 9: Character Customization

The user is then prompted to take a personality test to get one of 12 jam flavor types. This indicates which of the 6 characters (who will soon be introduced!) the user will match with.

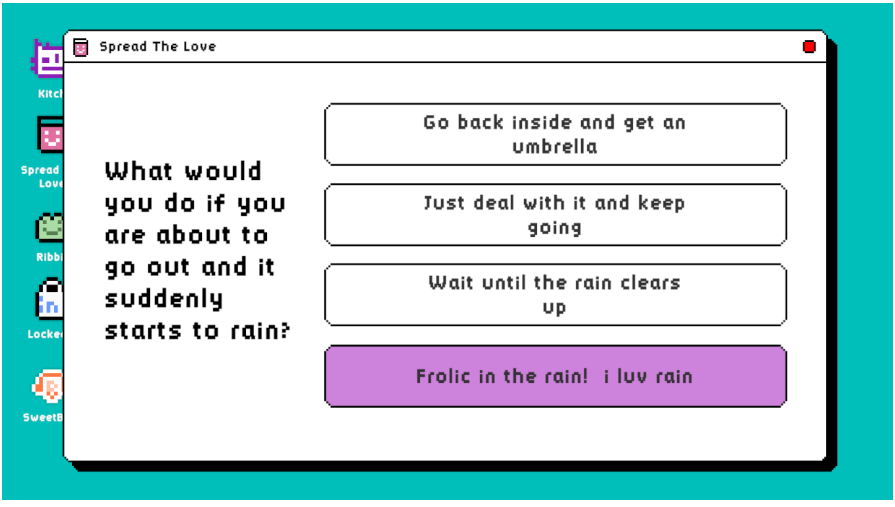


Figure 10: Personality Test

After the quiz, you're guided into creating a dating profile. This is where the off-kilter humor really shines. You are able to make your own bio, select (overly specific) likes and dislikes, and draw a beautiful signature.

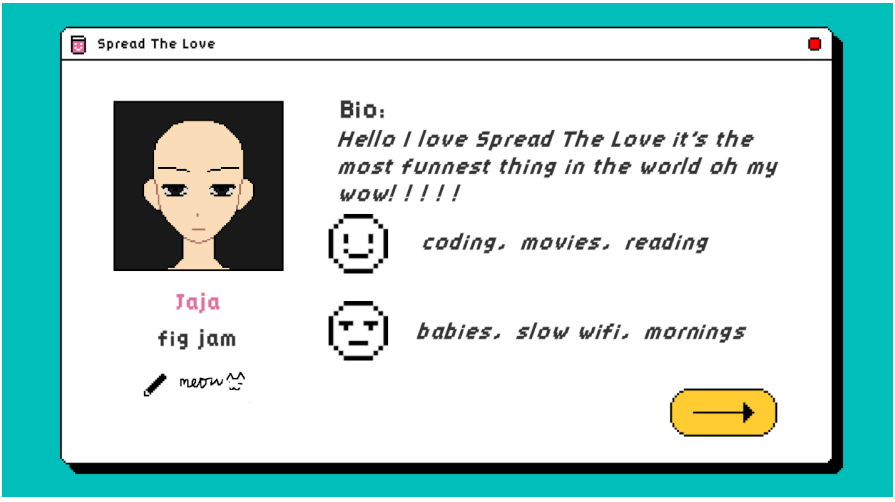


Figure 11: Player Profile

After profile creation, you are able to go through some potential dates and see our cast of 6 charming characters.

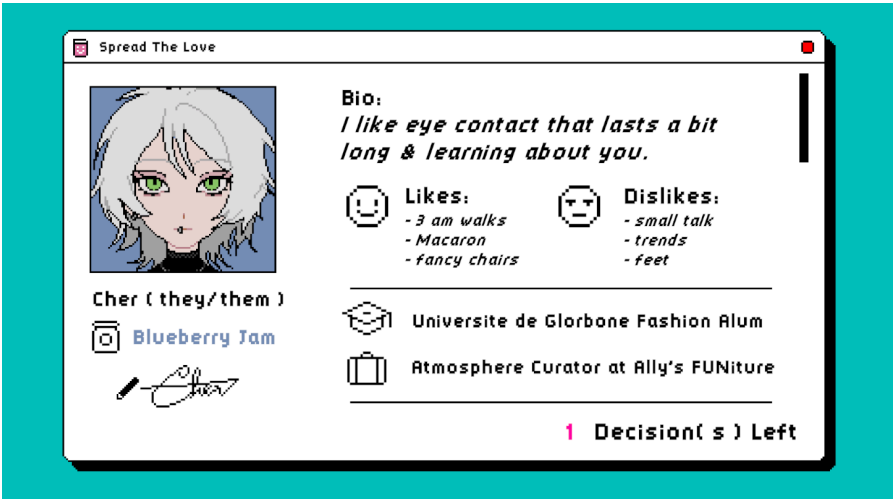


Figure 12: Character Profile

If you choose to pursue a match, you'll text with them and eventually go on a date, each capped off with a themed minigame to keep the pacing playful.

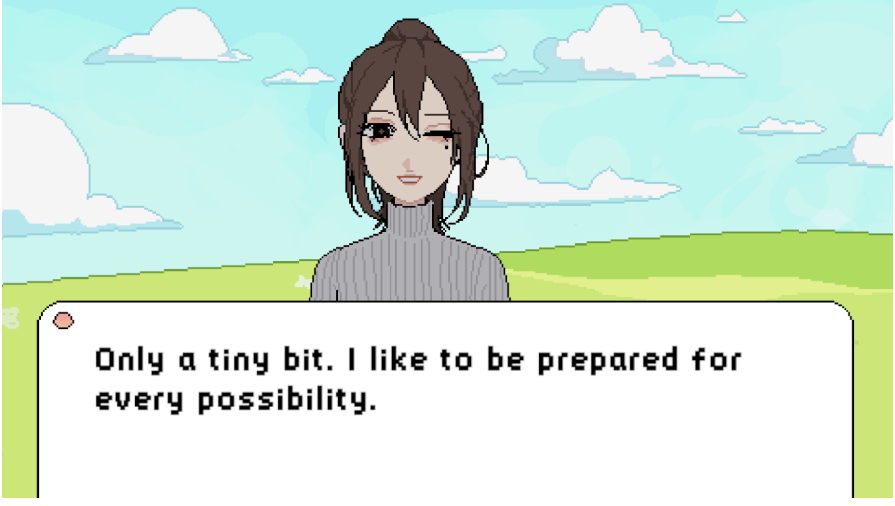


Figure 13: Date Dialogue

What Didn't Make the Cut

There were some fun cuts along the way.

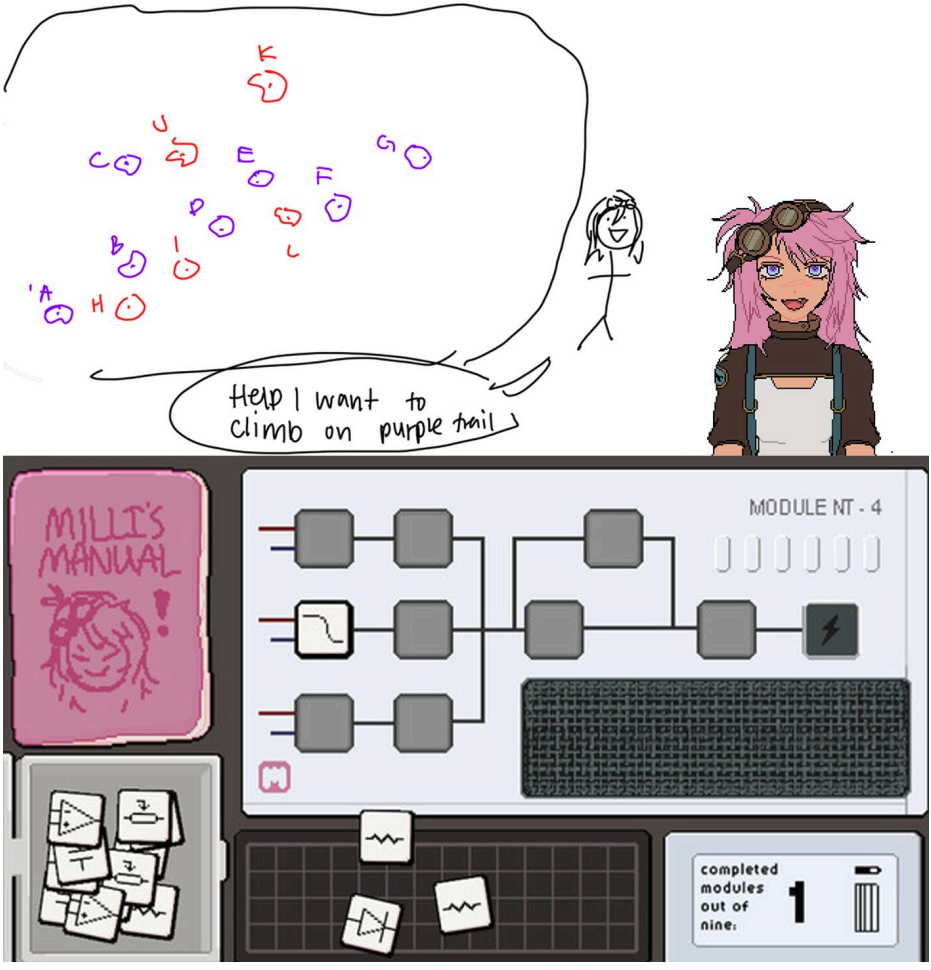


Figure 14: Milli's Minigame Mock-Up vs. Final

Milli's minigame was going to be a rock climbing game, but we couldn't make the perspective feel right, so it became an audio equalizer puzzle instead; this was actually inspired by my coursework as a computer engineering student.

Figure 15: Jet Design & Minigame Mock-Up vs. Final

Jet's minigame started as a full League of Legends parody ("Lingon Legends") but we pivoted toward a simpler, Street Fighter-style dupe to keep it more accessible.

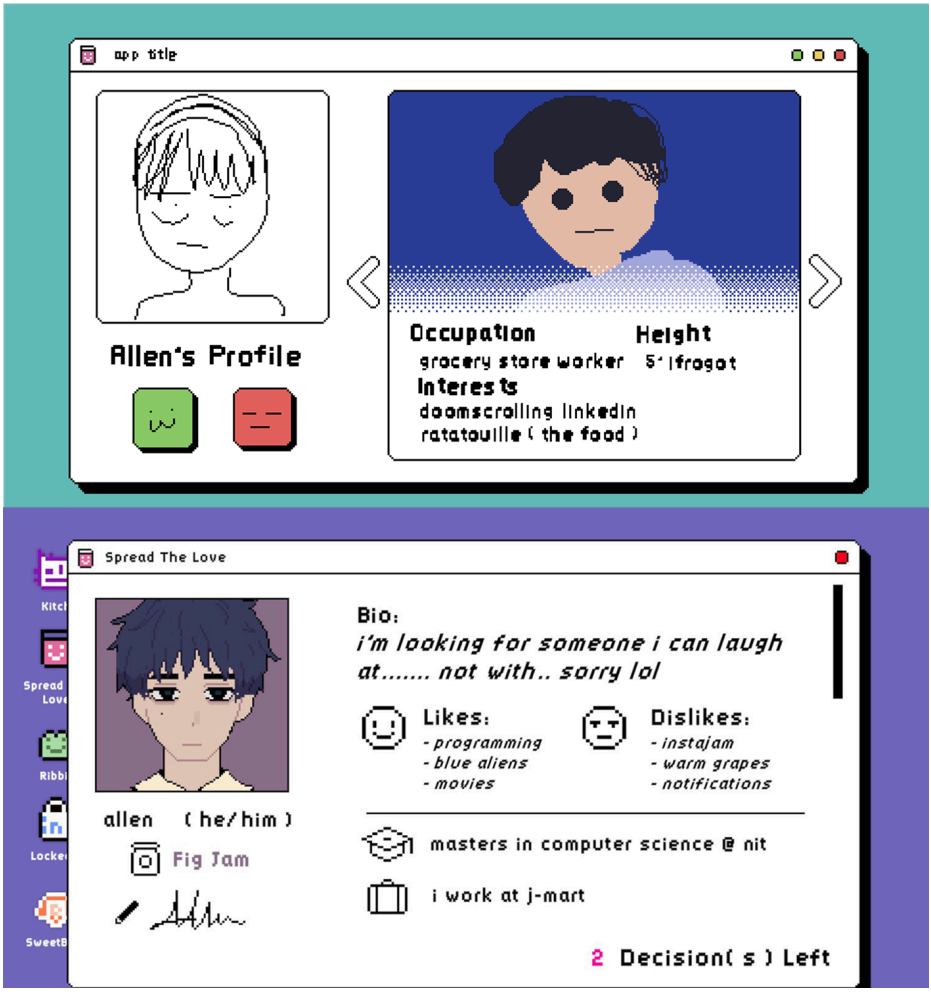


Figure 16: Dating Profile UI Mock-Up vs. Final

Dating profiles were supposed to have three photos each, but I didn't want to draw that many, so I reworked the UI to highlight text details instead.

Playtesting and Feedback

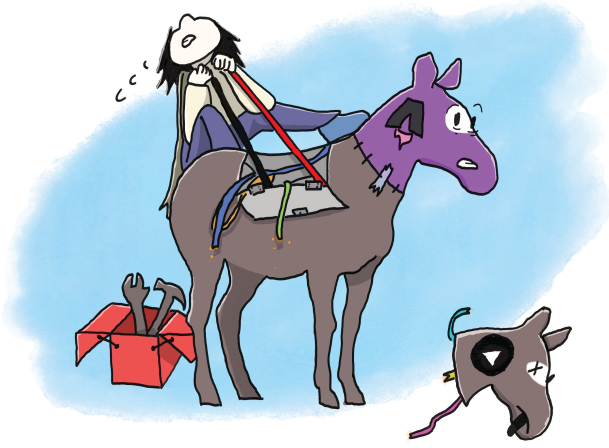
We had around 15–20 playtesters before release, mostly friends and people curious about our dev process. Watching them play was hilarious; everyone gravitated toward the character that matched their energy.

When we released the game on Steam, it wasn't perfectly polished, so we had to release a few patches and hotfixes based on feedback. But seeing players laugh, screenshot their results, and talk about their favorite dates made all the late nights worth it. These are still ongoing, and we are working on future patches and content updates.

Quick Learnings

Spending a summer on Spread the Love was like practicing "precise absurdity." We wanted the game to feel weird and self-aware, but also grounded enough to care about these pixel people.

It taught us how to focus on charm and tone, trim down unnecessary complexity, and ship something people genuinely enjoy. It also reminded me that cohesion is about meticulous planning, shared humor, and constantly clicking through every fake desktop window until it feels right.



The Great Events Site Migration

by Eric Park

In this paper-masquerading-as-a-research-paper-but-not-really-a-research-paper, I will discuss the process of migrating the Events site of Purdue Hackers from the antiquated NextJS-based codebase to a new AstroJS-based codebase. In the process, I'll also go over the process of integrating all the event metadata and details from Sanity, which we used to keep track of historical events, into the site codebase itself using the Content Collections feature of AstroJS.

Definitions

NextJS and **AstroJS** are two JavaScript (JS) frameworks that developers can use to build their webapps and websites. NextJS is primarily developed by the Vercel corporation, while AstroJS is built more by the community overall.

Sanity is a Content Management System (CMS), which ensures that "content" – in our case, past event information and retrospectives – all follow a specific format so that our frontend can easily convert the data coming over from Sanity into the final webpage that users can view. In addition, Sanity stores all the information in a database, along with the image assets associated with each event.

Finally, **TailwindCSS** is a CSS framework that allows web developers and designers to easily style the frontend (the part that users view) without having to maintain a separate CSS file. This is achieved by having almost all CSS functionality expressed as class names, which is included in the HTML markup.

Motivation

Purdue Hackers hosts several events throughout the academic year, including Hack Night, where creatives come together to work on projects and socialize. At midnight, a Checkpoint ceremony is held, where people present their projects and what they've been working on over the past two

weeks, and lots of photos are taken for posterity. Once the event is over, one of the organizers upload a postmortem of the event, including all the media taken during the event.

The initial version of our events site was developed by Matthew Stanciu, the past president of Purdue Hackers. Events were managed on Airtable, before the migration over to Sanity in January 2023 as Matthew wanted to use a real CMS to manage our events. For the RSVP functionality and emailing potential attendees, a GitHub Actions task ran that checked the RSVP email list hosted on Sanity and then sent out an email via a third-party service. We used Mailgun, before eventually switching over to Resend.

This system worked well for quite some time, but it wasn't without faults. The initial signs of trouble were reported by our very own organizers, who would use Sanity to write the postmortem to events. They would often report that Sanity was unreliable; it would lose uploaded image assets and force them to start over from scratch.

Additionally, because Sanity hosted our event data, each user interaction would require the server to query Sanity for the associated event information. A round trip between the browser and server backend would occur, the server would make another round trip to Sanity's servers, and then the response would then get sent over to the user. This increased the overall latency and responsiveness of the site, and required the backend to unnecessarily repeat the process of converting the data from Sanity into a list of events and the event detail page for the users. And as Sanity gave back the data in one giant payload, we had to use pagination to not cause undue strain on the overall infrastructure. Even the index page, with minimal information of just the title and date/time of the event, fetched the entire event metadata from Sanity, wasting a lot of users' data.

After a lengthy discussion on the engineering channel for Purdue Hackers, a solution was proposed: to statically generate the webpage, including all details about events in the codebase once, then serving the minified HTML to users. AstroJS had the promising feature of Content Collections that would allow us to achieve this goal, so it was our first pick out of the list of

alternatives to consider for this migration, which had been on our roadmap for quite some time.

The final nail in the coffin came when Ray Arayilakath, the current president of Purdue Hackers, transitioned the RSVP functionality over to Luma, a 3rd-party event and ticketing platform. Thus, current and future events were solely managed on this new platform, and the RSVP functionality and associated code became redundant on our events site codebase. We decided to take this opportunity to rewrite the codebase from scratch and base it off of AstroJS.

First Steps

On a separate branch, the first commit that wiped out our NextJS codebase[1] and set up a clean AstroJS template was created[2]. This marked the start of the migration attempt.

Before even looking into downloading and migrating the event metadata from Sanity, the initial structure of the events site was migrated by copying the HTML source from our NextJS codebase straight into the index page in our AstroJS codebase. After configuring the official TailwindCSS plugin[3], most of the styling displayed immediately with minor issues.

[1] https://github.com/purduehackers/events/commit/97217e07426cf092e889a7102354bb3fe4e5edc0

[2] https://github.com/purduehackers/events/commit/e0bd3b224ade828bb687d22a1abb8f733cae6af5

[3] https://docs.astro.build/en/guides/styling/#tailwind

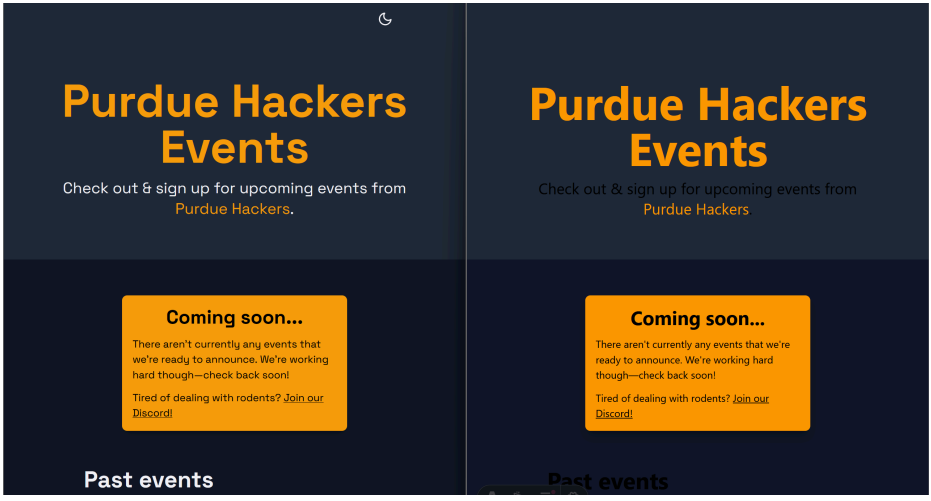


Figure 1: Left: the original events site with NextJS. Right: the initial AstroJS migration.

While mixing styling with markup might be a questionable decision for some, the choice of using TailwindCSS for our styling meant that migrating just the frontend to a new codebase became significantly easier, because the frontend's classes dictated how the content should be laid out on the page. Without having to set up a transpiler for separate CSS files, a plugin was all that was needed to have the page styling mimic the previous codebase.

One minor hurdle was that the old NextJS codebase utilized TailwindCSS v3, while the official plugin targeted TailwindCSS v4. During the migration, some of the configuration values defined in the dedicated `tailwind.config.js` file had to be moved over as CSS directives, like `@theme`. TailwindCSS's extensive library documentation helped immensely during this process, and I was able to match the original styling of the site.

Converting the events

The next stage was to preserve all of our old events and retrospectives. To achieve this, I had to download the event metadata from Sanity. Sanity however, does not use REST for their API endpoints. Instead, they have a

custom query language named GROQ[4] that I had to learn, just to query all the events that were stored in their backend.

For comparison, a typical SQL statement to query for data would look something like this:

```
SELECT * FROM events
WHERE date > 2020-03-24 AND date < 2025-01-01;
```

However, GROQ would require you to write:

```
*[_type == "event" && date > "2020-03-24" && date < "2025-01-01"]
```

Even though plain SQL could probably do most if not all of what GROQ achieves, it was what Sanity used, so it was what I had to learn in order to progress with the migration. Fortunately, I did not have to filter my result, as the goal was to grab everything I could off of Sanity. Thus, most of my experimentations with the GROQ query came down to how Sanity defined `_types` in their database.

For events metadata, the `_type` was of `event`, which was straightforward enough. However, for images stored within the event retrospective metadata, Sanity had a special `_type` of `sanity.imageAsset`. Each event metadata entry would store a collection of image asset IDs, which I would have to correlate with the `sanity.imageAsset` entry and then download from Sanity's servers, by constructing the full URL from that `imageAsset` entry.

Once the correct GROQ query was constructed, all the metadata could be downloaded with a single request. However, this did not include any of the images that were uploaded with the retrospectives, as mentioned previously. To facilitate this, several Python scripts were written[5] that handled the downloading, conversion, and renaming of all the events and images into the correct respective folders. This took several tries, mainly due to events with the same slug and names. In particular, Hack Nights without version

[4] https://www.sanity.io/docs/content-lake/how-queries-work

[5] https://github.com/purduehackers/events/tree/6e061709cc668f8c67cb586af6ede7211fce7b75/src/content

identifiers or "beta" Hack Nights that were held, confused the script and required modification.

But once the events were organized into each event category and the version-named subfolder, a single Content Collection configuration file[6] was all that was needed for AstroJS to correctly parse the schema and create a collection of events that could be used to query past events.

As mentioned earlier, AstroJS has a neat feature called "Content Collections" where you can define a schema in a configuration file. During compile time, Astro will look at this schema and determine all the files that fit within this schema with the glob pattern you have specified. If any files match the glob pattern but do not validate against the provided schema, a compile-time error is raised, making sure that all required data is accounted for in each event. This ensures consistency between all of our events metadata, while allowing us to track changes using Git commits.

Retrospective

Overall, the migration of the event site was a success, and once the PR was merged, a build job on Vercel ran and transparently replaced the old instance of our NextJS site with our new AstroJS instance, with zero downtime for users. Nearly all the functionality carried over, with only a handful of minor bugs[7] that escaped the testing phase of the PR before merging.

Through this experience, I learned that intermediary scripts, like the Python scripts we used to convert the events from the Sanity schema to AstroJS content collections, don't have to be perfect or pretty. Since they're designed to be run once and then discarded, the core objective is that they work, and in this instance, they've clearly served their purpose.

That extends to styling libraries like TailwindCSS. When I initially approached this library, I was part of the skeptics that thought mixing styling with the markup wouldn't work too well. However, once the frontend is written, we typically do not touch the markup and just import

[6] https://github.com/purduehackers/events/blob/main/src/content.config.ts
[7] https://github.com/purduehackers/events/issues/97

the component with the necessary data to display it to the user, which means the only time we will interact with the source is if we need to tweak the design, or migrate it like I've done here. And when you're doing either of those tasks, viewing both the styling and the markup is required, reducing the concern of combining the two.

Future Plans

While the migration itself was successful, maintaining the site will continue until we no longer need it or migrate off to something new. We already have a couple of ideas planned for the rewritten events site, most notably a redesign that will allow us to test out ideas for our upcoming overall brand renewal. As the codebase has been cleaned up, testing out new changes should be comparatively easy as we no longer have to account for features that we no longer use, such as the RSVP capabilities of the old events site.

After the migrated site launched, we received feedback that submitting new events and retrospectives through GitHub pull-requests add significant friction. This may come as ironic, given that I've just talked about the benefits that Astro's static content collections bring, but in the future, we may look into dynamically storing events metadata on a database, which would allow us to design a clean, friendly administrative interface that organizers can use to submit event details.

Another issue that cropped up was that, as it currently stands, our events site repository sits at nearly 2 GB of space used once cloned. This is due to all the image assets that we include with each event retrospective. For comparison, my personal website which was also built atop AstroJS didn't run into this issue with only a handful of images per post, if any. On our events site, each of our events retrospectives can contain around 20 to 50 images at once, which will not scale due to asset size. This is another area we could improve in, by perhaps storing images in a platform that's more suitable for the task, such as a CDN.

All in all, the rewrite has given us a solid foundation to improve our events site and to try out new things before they are propagated to the rest of our infrastructure.

Acknowledgments

Finally, many thanks to the various Purdue Hacker organizers and members for giving me feedback and words of encouragement during the migration phase, as well as bug reports that I didn't manage to catch pre- and post-deployment. I would also like to thank Kartavya for organizing this SIGHORSE initiative and for giving me a chance to write this paper.

A virtual summer art gallery in the form of a 3D cube

Prisha Bangera
Purdue University
Instagram: @prishainabox[1]

YouTube Demo Video[2]
Website Link[3]
Internet Archive Link[4]

When you open this interactive website, you are met with a 3D cube floating in space. Tiny stars orbit around the void. As you rotate around and zoom in with your mouse, you can peruse all the little digital artworks on the cube.

On this one cube sits three months of work: traditional art, digital art, and creative coding. In this article, I will explain the artworks displayed in the

[1] https://www.instagram.com/prishainabox/

[2] https://youtu.be/6x8JNF1w_sA

[3] https://prishasbangera.github.io/Virtual-Summer-Art-Gallery-2025/

[4] https://web.archive.org/web/20251014202937/https://prishasbangera.github.io/Virtual-Summer-Art-Gallery-2025/

gallery, how the project came to be, highlight future work, and weave in dashes of introspection.

Months of Art

When I first heard of SIGHORSE's existence, I was excited but not sure where to start. Three months later, and all I had was a pile of art made for various reasons. Each and every motivation behind why I created a piece of artwork represents a different aspect of the virtual art gallery. Overall, I can group these motivations into three main reasons.

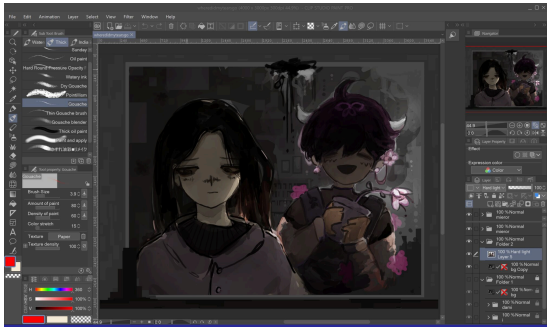
The first reason was something called Art Fight—an annual online art gifting game. I doodled my own original characters and created art from the characters of others. It was exciting to receive art from my online and real-life friends, as well as give art back in return as an "attack." So, this aspect of the gallery represents the stories of my characters and how I interpreted and shared those of others.

Shown here is one of my original characters on Art Fight—and one of the artworks in this cube gallery:
Dami. You can see my best friend's cousin's impossibly amazing version of my character in this Instagram post (@artlinxin)[5].

The second reason was impulse—perhaps a little spite as a CS major who just finished their second semester. I would suddenly start sketching or drawing, get an idea, experiment, maybe code, and (perhaps) complete the

[5] `https://www.instagram.com/p/DNEFRtQxpB9/?img_index=4`

piece. Unfortunately, there were a lot of works that I did not complete, so those were not included in this cube gallery project. Overall, though, this aspect of the gallery displays the impromptu and inconsistent nature of my art.



One of my unfinished artworks features Dami and another original character. This work is supposed to be paired with another artwork in this gallery. Can you see which one? Hopefully, I can finish it and add it to another cube gallery.

The final reason is the most extensive: at Purdue, I am part of Special Interest Group in Game Development (SIGGD), just like how SIGHORSE is the Special Interest Group in HORSEing around. SIGGD works together to create one game throughout one year—programming, assets, and all. I am part of the Programming and Art teams. For the latter, I focused mainly on environmental background art for the game.

The gallery contains some environmental art I created this summer for our game, Echoes of Isovios: A Legacy Undone[6]. However, I completed this world's background (Oblivion) before the summer began, and thus it is not allowed in the gallery!

For me, this part of the cube gallery represents my progress in digital art, composition, and design as I worked on the game. It also represents the largest project I have ever worked on with many other talented people.

The Project Idea

So, it was almost the start of the school year, and all I had was this random pile of art. It was only natural that I should somehow incorporate it into a final project. But how?

Initially, I wanted to make a realistic but virtual art gallery in which a user could amble about and explore. However, due to reasons,[7] I decided to opt for a singular cube base.

I ended up liking this design a lot better for a few reasons. Since the gallery is virtual, we can throw "realism" out the window. We don't have to worry about the user getting lost. Also, I later realized that this cube aspect has a

[6] `https://siggd.itch.io/siggd-game-2024-2025`

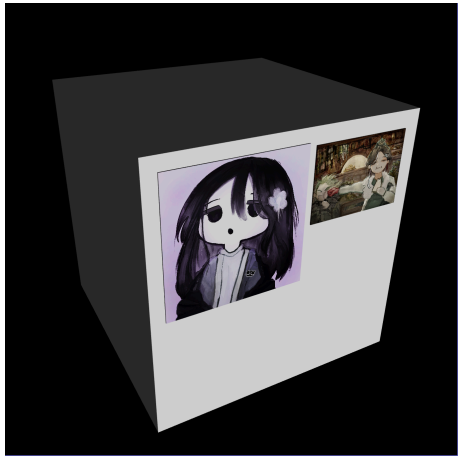
[7] Time constraints.

nice connection to my Instagram art account username, @prishainabox[8].

Creation Process

I decided to use p5.js to complete the project. p5.js is a JavaScript library which provides many tools for creating graphics in the browser. It also has a 3D rendering mode which drastically simplifies creation and interactivity. In fact, the base of the art gallery, a simple cube, only needed one simple function: `box`.

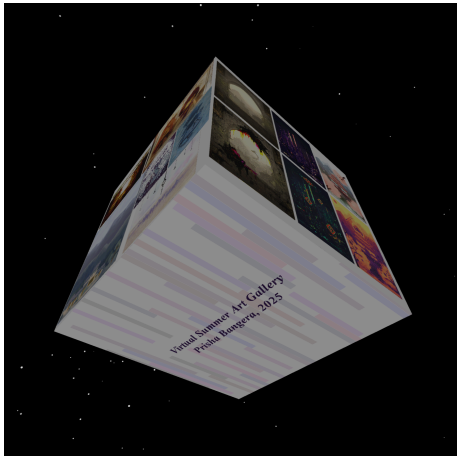
First, I gathered all the images I wanted to adorn the cube with. I placed them on the cube one by one. It took a while since every image had a different resolution and orientation.



The base of the art gallery, a plain cube, and two artworks already placed on it. Initially, the cube was much smaller than in the final project.

I also added a tiny animation: the artworks subtly go in and out throughout time. The title is located on the bottom of the cube, and I added animated translucent rectangles there as well.

[8] `https://www.instagram.com/prishainabox/`



The title is on the bottom of the cube, with translucent rectangles moving slowly up and down.

In a last minute idea, I also scattered some 3D stars around the space. While some of the stars are stationary, others are animated to orbit as time passes. At last, the black void outside of the gallery was not as empty anymore.

A view of the stars that orbit and surround the gallery.

Result

You can view the gallery here[9] and the code on GitHub[10]. Spoiler alert, though: I hope it does not crash your browser. If it does, there is a video link at the beginning of this article.

Snapshots of the final virtual art gallery.

As a personal art portfolio, it is fast (once it loads) and easy for the user to interact with.

Overall, I love how this project showcases all the artwork I completed over the summer. Even within this short time span, I improved my art by a lot– and the gallery shows this.

Future Work

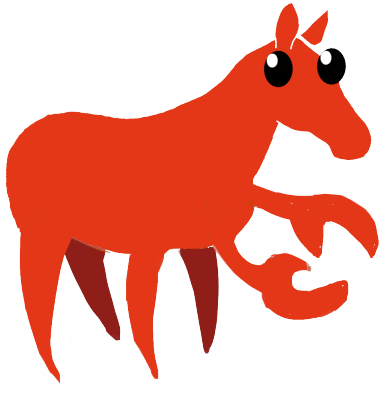
Besides refactoring the code of a very hastily made project, there are a few ideas for improvement. For one, I can move the whole project off p5.js and learn how to build it from scratch, rather than relying on predefined functions. By doing so, I can control more aspects like the camera and lights. Hopefully, I can also get feedback to improve the project's performance, loading time, and interactivity.

[9] `https://prishasbangera.github.io/Virtual-Summer-Art-Gallery-2025/`
[10] `https://github.com/prishasbangera/Virtual-Summer-Art-Gallery-2025`

Second, a quick search easily reveals the existence of rectangle packing algorithms. Instead of manually placing the portraits and artworks, perhaps there is a way to place all of the artworks automatically. A successful implementation could open the door for an application in which users can upload their own images and explore them using the cube gallery.

Final Remarks

I am happy with the way this project turned out, despite the room for improvement. Overall, the virtual art gallery cube effectively incorporates all the artwork I created over the summer–with or without reason. In extension, it includes the memories I made and the people I made them with.



The Generativity Pattern in Rust

Arhan Chaudhary

Abstract. The *generativity pattern* in Rust is a combination of typestate [1] and GhostCell [2], techniques that move what you'd normally check at run-time to compile-time. This pattern is not commonplace; its usage warrants a specific set of circumstances. However, it is a hugely important part of garbage [3] collection [4] utilities and other niche Rust crates. Aside from thinly spread academic literature[1], I haven't found an accessible analysis of this pattern online. In order to build up a full picture of the "what" and more importantly the "why," we will first spend some time walking through a realistic example to gauge the type of problem the generativity pattern solves—statically requiring data to come from or refer to the same source—as a *stronger* form of ownership. Then, we will introduce the generativity pattern and explain how to use it in the latter half of this article. Finally, we will follow up with a study of Crystal Durham [5]'s `generativity` [6] crate, a novel improvement to the generativity pattern.

Contents

[1] Yes, I will eventually get to them. You just need to keep reading.

1. Background

1.1. Permutations

Let us take the role of a crate author about permutations. We want to investigate the composition [7] of zero-indexed permutations. This can be expressed nicely visually.

Permutation composition

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

The permutation `b` defines the remapping of the elements from permutation `a`. Pretty simple. Notice that permutation composition is only possible under the following three conditions:

1. `a` and `b` must have the same length.
2. Every element from `a` and `b` must be non-negative and less than the length.
3. Every element from `a` and `b` must be unique.

Our library is general-purpose, so it is important to handle these error cases. Here is the simplest way to do that.

```
/// We provide a `compose_into` function in case the caller already
/// has a permutation preallocated. (This is good practice IMO).
pub fn compose_into(a: &[usize], b: &[usize], result: &mut [usize]) ->
Result<(), &'static str> {
    if a.len() != b.len() || b.len() != result.len() {
        return Err("Permutations must have the same length");
    }
    let mut seen_b = vec![false; a.len()];
    let mut seen_a = vec![false; b.len()];
    for (result_value, &b_value) in result.iter_mut().zip(b) {
```

```
        if *seen_b
            .get(b_value)
            .ok_or("B contains an element greater than or equal to the
            length")?
        {
            return Err("B contains repeated elements");
        }
        seen_b[b_value] = true;

        let a_value = a[b_value];
        if *seen_a
            .get(a_value)
            .ok_or("A contains an element greater than or equal to the
            length")?
        {
            return Err("A contains repeated elements");
        }
        seen_a[a_value] = true;

        *result_value = a_value;
    }
    Ok(())
}
```

Good on you if this made your Rust senses tingle because we shouldn't have to validate `a` and `b` every time. Rust allows us to enforce at the type level that they are valid permutations, using the newtype [8] design pattern.

```
pub struct Permutation(Box<[usize]>);

impl Permutation {
    pub fn from_mapping(mapping: Vec<usize>) -> Result<Self, &'static str> {
        // This function errors if `mapping` is an invalid
        // permutation or its length does not match the second
        // argument. The implementation is ommitted.
        validate_permutation(&mapping, mapping.len())?;
        Ok(Self(mapping.into_boxed_slice()))
    }

    pub fn compose_into(&self, b: &Self, result: &mut Self) -> Result<(),
    &'static str> {
        if self.0.len() != b.0.len() || b.0.len() != result.0.len() {
```

```
            return Err("Permutations must have the same length");
        }
        for (result_value, &b_value) in result.0.iter_mut().zip(&b.0) {
            // SAFETY: `b` is guaranteed to be a valid permutation
            // whose elements can index `self`
            *result_value = unsafe { *self.0.get_unchecked(b_value) };
        }
        Ok(())
    }

    pub fn compose(&self, b: &Self) -> Result<Self, &'static str> {
        let mut result = Self(vec![0; self.0.len()].into_boxed_slice());
        self.compose_into(b, &mut result)?;
        Ok(result)
    }
}
```

> Unsafe is going to be a recurring theme here. You've had your fair warning.

The newtype pattern is more useful than just for getting around the orphan rule. We restrict construction of `Permutation` to `Permutation::from_mapping`, which returns an error if the input is not a valid permutation. That means if we have an instance of `Permutation`, we don't have to worry about its mapping being potentially invalid, reducing the validation overhead to a length check and permitting unsafe during permutation composition. Rustaceans describe type-level guarantees like this by saying an *invariant* of `Permutation` is that it represents a valid permutation. Composing two permutations upholds this invariant, so we expose `Permutation::compose` to create a new `Permutation` from existing ones.

This code is a major improvement! It is simple, easy to use, and it provides reasonable errors. However, a closer examination reveals some problems:

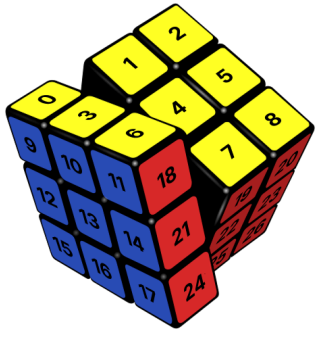
- Every call to our composition function spends time performing a length check. Our example is simplistic so it happens to be cheap, but this type of

check may require more expensive logic in a practical scenario. Note that we can't use const generic lengths because our library operates on arbitrarily-sized slices at run-time.
- Returning a `Result` forces the caller to be prepared to handle the error variant. Library users might be able to guarantee that the length checks will pass, which would make the error handling more annoying than helpful.

Yes, these aren't *important* problems per se, but they are still inconveniences to be aware of.

1.2. Permutation groups

We now want to extend our library to model a permutation group [9], a description of a set of permutations. In a permutation group, every permutation in the set can be written as a sequence of compositions of a select few *base permutations*, which we will use to represent the entire collection. For example, the manipulations of the Rubik's Cube form a permutation group. Its base permutations which represent the entire permutation group are the six face rotations. By definition, every possible state on the Rubik's Cube can be reached from a combination of those face rotations.



The illustrated turn is a permutation fifty-four elements long, because there

are fifty-four stickers on a Rubik's Cube[2].

It follows that if you compose two permutations in a permutation group, the resulting permutation will also be a permutation in that group. The reasoning is not so relevant; take this at face value.

A reasonable data structure for permutation groups looks like this:

```
pub struct PermGroup {
    base_permutation_length: usize,
    base_permutations: Vec<Permutation>,
}

impl PermGroup {
    pub fn new(
        base_permutation_length: usize,
        base_permutation_mappings: Vec<Vec<usize>>,
    ) -> Result<Self, &'static str> {
        for mapping in &base_permutation_mappings {
            validate_permutation(mapping, base_permutation_length)?;
        }
        Ok(Self {
            base_permutation_length,
            base_permutations: base_permutation_mappings
                .into_iter()
                .map(|mapping| Permutation(mapping.into_boxed_slice()))
                .collect::<Result<Vec<Permutation>, &'static str>>()?,
        })
    }

    pub fn base_permutations(&self) -> &[Permutation] {
        &self.base_permutations
    }
}
```

Your *inner Ferris* awakens. With the annoyances of our last iteration freshly in memory, you ask yourself: can we perform that length check (the `validate_permutation` function) during the creation of `PermGroup`, and avoid it entirely in `Permutation::compose_into`? Then, can we tweak our

[2]The center stickers don't actually move, and thus can be ignored, so the illustrated turn is traditionally simplified to a permutation forty-eight elements long.

composition function to only operate on permutations from the same permutation group?

```
impl Permutation {
    // No `from_mapping` method. `Permutation` can only be
    // constructed within `PermGroup::new`.

    pub fn compose_into(&self, b: &Permutation, result: &mut Permutation) {
        for i in 0..result.0.len() {
            // SAFETY: ... ?
            unsafe {
                *result.0.get_unchecked_mut(i) =
                *self.0.get_unchecked(*b.0.get_unchecked(i));
            }
        }
    }

    pub fn compose(&self, b: &Permutation) -> Permutation {
        let mut result = Self(vec![0; self.0.len()].into_boxed_slice());
        self.compose_into(b, &mut result);
        result
    }
}
```

All of a sudden, we've opened up an unsafety hole! We implicitly assumed that the permutations to compose were from the same permutation group. This is not necessarily true: what if a library user composes two base permutations from different permutation groups? If the permutation lengths *don't* match, `get_unchecked` will index out of bounds and exhibit undefined behavior; this is clearly a problem! The intent of this operation is obviously nonsensical, but it does not change the fact that it is still our responsibility, as the crate author, that the safe functions we provide can never cause undefined behavior.

There is a more fundamental reason to care about this unsoundness if left unchecked. An invariant of permutation composition within the same permutation group is membership; if the permutations to compose are in the same permutation group, the resulting permutation is also in that group. Even if the lengths of permutations from two different permutation groups

did match, composing them could produce a permutation outside of either group, which is a logic error. Other code may even have unsafe blocks that rely on permutation group membership, for example a Rubik's Cube solver optimized for speed.

Mitigating this by checking permutation group membership every function call is a very expensive operation. This is an example of the "practical scenario" mentioned beforehand.

We have demonstrated that the newtype pattern alone is not powerful enough to prevent this logic error. We will analyze different approaches that ensure our library only permits permutation composition within the same permutation group. Each has their own trade-offs, but are all right answers for different situations. They will also lay the groundwork to justify using the generativity pattern.

All the code segments provided in this article can be found here [10].

2. The unsafe approach

The simplest solution is to mark `Permutation::compose_into` and `Permutation::compose` unsafe.

```
/// # Safety
///
/// `self`, `b`, and `result` must all be from the same
/// permutation group.
pub unsafe fn compose_into(&self, b: &Permutation, result: &mut Permutation) {
    for i in 0..result.0.len() {
        // SAFETY: permutations within the same group can be
        // composed.
        unsafe {
            *result.0.get_unchecked_mut(i) =
            *self.0.get_unchecked(*b.0.get_unchecked(i));
        }
    }
}
```

Although the extent of the undefined behavior with permutation composition is just the bounds checking, the goal of this approach is to

enforce permutation group membership. Thus, the above safety contract is made more restrictive to reflect this idea. The usage of unsafe to maintain a validity invariant is contentious. Permutation composition of the same length within different permutation groups is a logic error, and it violates the safety contract, but is not technically unsafe. Sure, you might panic later on or get some other issue, but this alone will never cause undefined behavior.

To play devil's advocate, since we only care about composition within the same permutation group, one may consider producing an invalid value from this type of permutation composition undefined behavior. With the safety contract's additional restriction, calling code no longer has to worry about handling this logic error, while additionally gaining the contextual benefit of this assumption. Personally, I believe this use of unsafe is warranted—at the end of the day, the safety contract does still prevent undefined behavior. I encourage you to have your own opinion [11].

If you don't care about using unsafe—and there are valid reasons not to—then this might be what you want. That said, it's not always going to be this simple. What if we introduce a new trait, `ComposablePermutation`, that generalizes over different permutation representations? For example, the `PSHUFB` instruction can compose two permutations in a single clock cycle if they have less than sixteen elements.

```
pub trait ComposablePermutation: Clone {
    fn from_mapping(mapping: Vec<usize>) -> Result<Self, &'static str>;

    /// # Safety
    ///
    /// `self`, `b`, and `result` must all be from the same
    // permutation group.
    unsafe fn compose_into(&self, b: &Self, result: &mut Self);

    /// # Safety
    ///
    /// `self` and `b` must both be from the same permutation
    // group.
    unsafe fn compose(&self, b: &Self) -> Self {
        let mut result = self.clone();
```

```
        // SAFETY: `self`, `b`, and `result` are all from the
        // same permutation group.
        unsafe { self.compose_into(b, &mut result) };
        result
    }
}

impl ComposablePermutation for Permutation {
    // ...
}
```

The consequences of using unsafe begin to show. Because our generic `Permutation` implements `ComposablePermutation`, and we have shown that permutation composition from different permutation groups may cause undefined behavior, `Permutation::compose_into` must be made unsafe at the trait level. Rust doesn't allow us to only make `Permutation`'s implementation unsafe. Either all implementers must be made unsafe, or none at all. In a library *about* permutation composition, we have now forced our users to wrangle with unsafe for its most essential operation. Not just with `Permutation::compose_into`, but with all of their own implementations of `ComposablePermutation`!

"That is completely unfair!" You might say. *"This is a small edge condition I don't care about. I'm going to mark this trait method safe anyways."* Well, the Rust community generally has a zero-tolerance stance on undefined behavior; the last time someone wanted to mark an unsound method safe, it didn't end very well [12].

3. The atomic ID approach

The second approach is to validate our base permutations *upfront* and use a private integer to associate them to a unique permutation group. This simplifies the test for permutation group membership to a cheap integer comparison. Internalizing how this approach works will be crucial to understanding the generativity approach. Rereading is encouraged.

```rust
use std::sync::atomic::{AtomicU64, Ordering::Relaxed};

pub struct PermGroup {
    base_permutation_length: usize,
    base_permutations: Vec<Permutation>,
    id: u64,
}

static ID: AtomicU64 = AtomicU64::new(0);

impl PermGroup {
    pub fn new(
        base_permutation_length: usize,
        base_permutation_mappings: Vec<Vec<usize>>,
    ) -> Result<Self, &'static str> {
        for mapping in &base_permutation_mappings {
            validate_permutation(mapping, base_permutation_length)?;
        }
        let id = ID.fetch_add(1, Relaxed);
        Ok(Self {
            base_permutation_length,
            base_permutations: base_permutation_mappings
                .into_iter()
                .map(|mapping| Permutation(mapping.into_boxed_slice(), id))
                .collect(),
            id,
        })
    }

    pub fn base_permutations(&self) -> &[Permutation] {
        &self.base_permutations
    }
}
```

The implementation of `PermGroup` does not actually change much. As before, we check that all mappings from `base_permutation_mappings` are valid permutations of the same length before creating a new `PermGroup`. This time, we utilize a global `AtomicU64` to uniquely identify the permutations in a permutation group, passing it as an integer to `Permutation`. The integer is guaranteed to be unique for `Permutations` among different `PermGroups` because we increment the identifier every call to `PermGroup::new`.

```rust
pub struct Permutation(Box<[usize]>, u64);

impl Permutation {
    pub fn from_mapping_and_group(
        mapping: Vec<usize>,
        group: &PermGroup,
    ) -> Result<Self, &'static str> {
        validate_permutation(&mapping, group.base_permutation_length)?;
        let permutation = Self(mapping.into_boxed_slice(), group.id);
        validate_permutation_group_membership(&permutation,
        &group.base_permutations)?;
        Ok(permutation)
    }

    pub fn compose_into(&self, b: &Self, result: &mut Self) -> Result<(),
    &'static str> {
        if self.1 != b.1 || b.1 != result.1 {
            return Err("Permutations must come from the same permutation
            group");
        }
        for i in 0..result.0.len() {
            // SAFETY: `self`, `b`, and `result` have the same ID.
            // Therefore, they are members of the same group and
            // can be composed.
            unsafe {
                *result.0.get_unchecked_mut(i) =
                *self.0.get_unchecked(*b.0.get_unchecked(i));
            }
        }
        Ok(())
    }

    pub fn compose(&self, b: &Self) -> Result<Self, &'static str> {
        let mut result = Self(vec![0; self.0.len()].into_boxed_slice(),
        self.1);
        self.compose_into(b, &mut result)?;
        Ok(result)
    }
}
```

Creating a new `Permutation` now requires a mapping and a `PermGroup` reference. Once the mapping is verified as both a valid permutation and a member of that permutation group, only then is a new `Permutation` created

with that `PermGroup`'s identifier, as a "token" of its membership. We can no longer create `Permutations` willy-nilly from just a mapping because that would offer no guarantees about the uniqueness of its identifier.

The fruits of our labor are rewarded in `Permutation::compose_into`. The expensive permutation group membership test is performed exclusively during `Permutation`'s creation. When two permutations are composed, those same "tokens" are used to cheaply verify membership within the same permutation group. Hence, callers can safely assume permutation composition produces another permutation in the same permutation group without compromising efficiency.

This solution is likely to be considered good enough in industry—most practitioners would need a good reason to care more about this problem. However, if your interest is piqued, what would really be nice is an *infallible* yet *zero-cost* permutation composition operation—one that is guaranteed to be valid at compile-time and as fast as the unsafe approach. If you're willing to go a small step farther, we arrive at…

4. The generativity approach

The big reveal: the generativity approach is equivalent to the atomic ID approach, except everything is done at compile-time. Generativity solves the fundamental problem thus far: the invariant of `Permutation` guarantees it is a valid permutation, but not that it is necessarily associated with a specific `PermGroup`.

Existing literature achieves generativity by sacrificing ergonomics and readability. They require wrapping all code in (often deeply nested) closures, warding off much of their interest in practice. We will spend the rest of this article examining Crystal Durham [5]'s `generativity` [6] crate, which utilizes a novel and highly experimental technique to achieve generativity without needing a closure. Later, we will show that the `generativity` crate is a zero-cost compile-time abstraction.

(Subtle-but-no-so-subtle foreshadowing: we will explore my own improvement to this technique in the next section)

```
use generativity::{Guard, Id, make_guard}

fn main() {
    // Create a variable `guard` of type `Guard<'_>`
    make_guard!(guard);
    // Consume that `guard` into an `Id<'_>`
    let id: Id<'_> = guard.into();
}
```

`generativity` publicizes three things: `Guard`, `Id`, and `make_guard`. Invoking the `make_guard` macro creates a `Guard<'_>` with a let binding, an identifier that carries a guaranteed unique lifetime. This lifetime is not actually used as a lifetime in the usual sense. It exists solely to make each instance of `Guard<'_>` a unique type. This is not voluntary nor merely a suggestion; the following *does not compile* because `make_guard`'s lifetime uniqueness guarantee cannot be broken[3].

```
fn unify_lifetimes<'a>(a: &Guard<'a>, b: &Guard<'a>) {}

make_guard!(a);
make_guard!(b);
// rejected: unique lifetimes cannot be unified (this
// is not the actual compiler error message)
unify_lifetimes(&a, &b);
```

`Id<'_>` is like `Guard<'_>` except that it implements `Copy` and `Clone` while the latter does not. So, to create distributable copies of this identifier, you must consume a `Guard<'_>` into an `Id<'_>` using its `From` implementation. This is all that `generativity` exports.

[3]This lifetime is not technically unique. You could unify it with another lifetime in a similar function call: `fn unify_lifetimes<'id>(impostor: &'id (), guard: &Guard<'id>) { ... }`. The lifetime is only unique among the provided `Id<'id>` and `Guard<'id>` types, so as long as your code only trusts lifetimes carried by those types it will be sound.

With the generativity pattern in place, the body of `PermGroup::new` remains sound because it creates `Permutations` with the same lifetime identifier, making it unique among different `PermGroups`.

```
pub struct PermGroup {
pub struct PermGroup<'id> {
    base_permutation_length: usize,
    base_permutations: Vec<Permutation>,
    id: u64,
    id: Id<'id>
}
```

```
pub fn new(
    base_permutation_length: usize,
    base_permutation_mappings: Vec<Vec<usize>>,
    guard: Guard<'id>,
) -> Result<Self, &'static str> {
    for mapping in &base_permutation_mappings {
        validate_permutation(mapping, base_permutation_length)?;
    }
    let id = ID.fetch_add(1, Relaxed);
    let id = guard.into();
    Ok(Self {
        base_permutation_length,
        base_permutations: base_permutation_mappings
            .into_iter()
            .map(|mapping| Permutation(mapping.into_boxed_slice(), id))
            .collect(),
        id,
    })
}
```

Why is `guard` passed as an argument, and why isn't `make_guard` creating it within the function body? This reveals `generativity`'s implementation caveat: a `Guard<'id>` can never escape the scope it was defined in. Think of creating a `Guard<'id>` as creating a reference to a local variable. No matter what, it is only valid inside of its enclosing scope.

So, instantiating two different permutation groups, for example, looks like this:

```
make_guard!(guard1);
make_guard!(guard2);
let first = PermGroup::new(..., guard1);
let second = PermGroup::new(..., guard2);
// rejected: `guard1` used after move
// let third = PermGroup::new(..., guard1);
```

The purpose of `Guard<'id>` when `Id<'id>` already exists becomes clear when considering that `third` is rejected by the compiler. If `PermGroup::new` accepted an `Id<'id>`, two different permutation groups could be assigned the same `Id<'id>` because it implements `Copy`.

Okay, this is all fine and dandy, but how does this help improve permutation composition?

```
pub struct Permutation(Box<[usize]>, u64);
pub struct Permutation<'id>(Box<[usize]>, Id<'id>);
```

Recall that every `Id<'id>` carries a unique lifetime among different `PermGroup<'id>`s. By combining `Permutation` with `Id<'id>` and a lifetime parameter, we create a collection `Permutation<'id>`s whose types are the same within a `PermGroup<'id>` but distinct from permutations within other `PermGroups<'id2>`s. Our permutation composition function takes in the same type `Self` for all arguments—it follows that `Permutation<'id>`s from different permutation groups cannot be composed as they are not the same type, and `Permutation<'id>`s from the same permutation group can be composed as they are the same type. This is the essence of the generativity pattern, enforced at compile-time.

Behold: a permutation composition function that is unchecked, infallible, and safe. The full implementation is here [13].

```
pub fn compose_into(&self, b: &Self, result: &mut Self) -> Result<(),
&'static str> {
    if self.1 != b.1 || b.1 != result.1 {
        return Err("Permutations must come from the same permutation group");
```

```
    }
pub fn compose_into(&self, b: &Self, result: &mut Self) {
    for i in 0..result.0.len() {
        // SAFETY: `self`, `b`, and `result` are members of the
        // same group and can be composed.
        unsafe {
            *result.0.get_unchecked_mut(i) =
            *self.0.get_unchecked(*b.0.get_unchecked(i));
        }
    }
    Ok(())
}
```

Let us informally prove `generativity`'s equivalence to the atomic ID approach:

- In the atomic ID approach, unique integer identifiers are created à la `ID.fetch_add(1, Relaxed)`. This directly parallels `make_guard!(guard)`, which creates a unique `Guard<'id>` identifier.
- The unique integer identifier is then stored inside a primitive `u64`. This implements `Copy` and it is distributed among the input base permutations to associate each one with its permutation group. Similarly, `Id<'id>` serves this purpose.
- The unique integer identifier is used to test permutation group membership during permutation composition, erroring if not the case. The generativity pattern directly embeds the same test into the type system.

It would be irresponsible for me to advertise the `generativity` crate as a perfect solution barring its implementation caveat. Yes, the implementation caveat is its only functional limitation, but there are some developer experience problems to consider.

- Although there's only a single line marked unsafe in our `Permutation<'id>` example, its soundness is now much harder to justify. It is on the developer to prove that `'id` uniquely associates permutations to their permutation group; any mishandling could easily make the whole thing unsound (i.e. if `PermGroup::new` took an `Id<'id>` instead of a `Guard<'id>`).

- The `'id` lifetime, like all other lifetimes, is pervasive. Data structures that wish to store `Guard<'id>` or `Id<'id>`, or any other data structure that stores `Guard<'id>` or `Id<'id>`, must have a lifetime annotation. But `make_guard` is typically invoked at the outermost scope, and will likely have to be passed through many types, so the number of lifetime annotations is amplified.
- Some APIs require types to satisfy `'static`. For example, the closure passed to `thread::spawn` must satisfy `'static`, making it impossible to return a `Permutation<'id>`. The workaround is usually an inconvenient band-aid; in this case the standard library offers `thread::scope` to borrow non-`'static` data in a thread.
- The compiler errors [14] from misusing `generativity` don't provide the location of the error. They are generally confusing, even when you know how `generativity` internally works.

The atomic ID approach shares only the first problem. Given your use case, it might be what you want.

4.1. The fundamental purpose

Notice how different instances of `Permutation<'id>` masquerade as separate types even though they have the same underlying data representation. In Rust, `'id` is known as a *branded lifetime*, or more generally, a *type brand*. The first principles of type branding date back at least to the work of John Launchbury and Simon Peyton Jones on the ST monad in Haskell [15] (section 2.5.2) in 1995. Aria Desires' master's thesis [16] (section 6.3) brought this into the context of Rust in 2015, coining lifetime branding in Rust with the term "generativity." The more recent GhostCell paper [2] by Joshua Yanovski and others utilized generativity to present interior mutability as a zero-cost abstraction in 2021.

This segues into an important point. The fundamental purpose of the generativity pattern is not necessarily to improve performance, but to statically require that individual values come from or refer to the same source. The performance benefits are a symptom of this requirement. I like to informally think of it as a stronger form of ownership.

For an alternative perspective, Aria Desires' master's thesis explores this idea with a concept called a `BrandedVec`. When the `i`th element of an ordinary vector `vec` is accessed through `&vec[i]`, a run-time check is performed to see if `i` is in bounds. If not, then the program will panic. However, in many situations we know that the indices are always in bounds; one such case regards the append-only vector, the `BrandedVec`. All elements pushed to this type of vector are forever valid. Leveraging this fact, the push operation returns the index of the pushed element so it can later be used to soundly perform unchecked indexing.

If we wanted to mark the unchecked indexing operation safe, this returned index can't be an ordinary `usize`. Bad actors may provide their own `usize` instead of one returned from the push operation. This returned index can't be a newtyped `usize` either. The problem with this is a microcosm of the problem with our initial `Permutation` example: different indices from different append-only vectors may be used to unsoundly index one another. To make our accesses safe, the solution is to lifetime brand the returned indices to statically associate them with `vec`—the generativity pattern.

As hinted to beforehand, generativity has traditionally only been achieved through the use of a closure. The GhostCell paper includes this following example, taken from its inspired adaption [17] of `BrandedVec`.

```
let vec1: Vec<u8> = vec![10, 11];
let vec2: Vec<u8> = vec![20, 21];

BrandedVec::new(vec1, move |mut bvec1: BrandedVec<u8>| {
    bvec1.push(12);
    let i1 = bvec1.push(13);
    let _idx = bvec1.get_index(0).unwrap();
    BrandedVec::new(vec2, move |mut bvec2: BrandedVec<u8>| {
        let i2 = bvec2.push(22);
        println!("{:?}", bvec2.get(i2)); // No bound check! Prints 22
        *bvec2.get_mut(i2) -= 1;         // No bound check!
        println!("{:?}", bvec2.get(i2)); // Prints 21
        println!("{:?}", bvec1.get(i1)); // Prints 13
        // rejected: i1 is not an index of bvec2
        // println!("{:?}", bvec2.get(i1));
```

```
    });
});
```

Each `BrandedVec` created from a `Vec` receives its own lifetime brand within each closure[4]. In terms of ergonomics, it's not exactly your Friendly Neighborhood Spider-Man.

But through the `generativity` crate, we eliminate the rightward drift, changing just nine lines [20] of `BrandedVec`'s implementation. The API now appears less foreign and more Rust-like.

```
let vec1: Vec<u8> = vec![10, 11];
let vec2: Vec<u8> = vec![20, 21];

make_guard!(guard1);
let mut bvec1 = BrandedVec::new(vec1, guard1);
bvec1.push(12)
let i1 = bvec1.push(13);
let _idx = bvec1.get_index(0).unwrap();

make_guard!(guard2);
let mut bvec2 = BrandedVec::new(vec2, guard2);
let i2 = bvec2.push(22);
println!("{:?}", bvec2.get(i2)); // No bound check! Prints 22
*bvec2.get_mut(i2) -= 1;         // No bound check!
println!("{:?}", bvec2.get(i2)); // Prints 21
println!("{:?}", bvec1.get(i1)); // Prints 13
// rejected: i1 is not an index of bvec2
```

[4]Rust doesn't have rank-2 polymorphism [18], so we need to replicate it using a closure with a Higher-Rank Trait Bound [19]. The type signature of the closure passed to `BrandedVec::new` is `inner: impl for<'id> FnOnce(BrandedVec<'id, T>) -> R`, and this just means every call to `inner` must be prepared to handle an argument with any possible lifetime. Within a single function the compiler has perfect information, but calling `inner` inside `BrandedVec::new` tricks the borrow checker. Since it doesn't (and will likely never) do interprocedural analysis, it conservatively sees every call to `inner` as producing an opaque lifetime that can't be unified with any other. To avoid any relation with an existing lifetime, a fresh new lifetime is statically generated for every call to `BrandedVec::new`, our lifetime brand for `BrandedVec<'id, T>`. This is just a brief overview of a well-investigated topic. Repeating for convenience, further reading is encouraged here [16] (section 6.3) and here [2] (section 2.2.1).

```
// println!("{:?}", bvec2.get(i1))
```

Our digression in bringing up this comparison has an ulterior motive. The original closure technique bears the exact same implementation caveat with `generativity`: nothing declared inside of the closure can escape it. `make_guard` effectively does the same thing as wrapping the rest of the function in an immediately invoked closure, and is no less capable than the closure technique.

4.2. Why the implementation caveat?

Let us offer another perspective as to why `Guard<'id>` cannot escape its defining scope. StackOverflow user rodrigo [21] points out [22] that you can achieve something similar to `generativity` using an anonymous unit struct and a macro to create the permutation group. Successive calls to this macro create permutation groups branded by this newly generated unit struct. In the context of our `Permutation` example, example usage looks like this. The full implementation is here [23].

```
#[macro_export]
macro_rules! new_perm_group {
    ($len:expr, $mappings:expr) => {{
        let len = $len;
        let mappings = $mappings;
        struct InvariantToken;
        // SAFETY: private API, only used in this macro.
        unsafe {
            $crate::PermGroup::<InvariantToken>::new(len, mappings)
        }
    }};
}
```

```
let first_perm_group = new_perm_group!(4, vec![vec![1, 2, 0, 3]]).unwrap();
let second_perm_group = new_perm_group!(3, vec![vec![2, 0, 1]]).unwrap();
let first_perm = &first_perm_group.base_permutations()[0];
let second_perm = &second_perm_group.base_permutations()[0];
```

```
// rejected: `first_perm` and `second_perm` are not the same type
// first_perm.compose(second_perm);
```

The flaw is quite subtle. The macro constructor creates a token per-*call-site* instead of per-*owner*. Every expression results in a particular type; if the same macro is run more than once, it will produce the same type, even if it is unique to the expression. This can be exploited to give multiple owners the same brand. To exemplify:

```
let first = (4, vec![vec![1, 2, 0, 3]]);
let second = (3, vec![vec![2, 0, 1]]);

let mut perm_groups = vec![];
for (len, mappings) in [first, second] {
    // I expanded the macro to make it easier to understand!
    // perm_groups.push(new_perm_group!(len, mappings).unwrap());

    perm_groups.push({
        let len = len;
        let mappings = mappings;
        struct InvariantToken;
        // SAFETY: private API, only used in this macro.
        unsafe {
            crate::PermGroup::<InvariantToken>::new(len, mappings)
        }
    }.unwrap());
}
let first_perm = &perm_groups[0].base_permutations()[0];
let second_perm = &perm_groups[1].base_permutations()[0];

// not rejected, UB!
first_perm.compose(second_perm);
```

We have just invoked undefined behavior from safe user-facing code. This is unsound without question, and there is no point in endorsing this approach.

There is a remedy: combine `InvariantToken` with a locally-scoped lifetime, as illustrated [24] in binarycat [25]'s crate `typetoken` [26]. This only creates a strictly less capable version of `generativity`. There is no point in endorsing this approach either.

If the above code were possible with `generativity`, `'id` could escape the scope and assign all elements of the vector the same lifetime brand. We would have the exact same unsoundness problem, thus we cannot use a loop to create a dynamic number of `PermGroup<'id>`s.

5. How does `generativity` work?

At this point a good part of my readers are *itching* to know what makes the `generativity` crate so magical compared to the age-old closure technique. The suspense is probably killing you. Or more likely putting you to sleep. We will introduce the inner workings of `generativity` top-down. I will first present my own minimal rewrite of the `generativity` crate, called `min_generativity`. Then, we will comprehensively walk through how each part of it works.

5.1. `min_generativity`

```
use std::marker::PhantomData;

pub type Id<'id> = PhantomData<fn(&'id ()) -> &'id ()>;

pub struct Guard<'id>(pub Id<'id>);

impl<'id> From<Guard<'id>> for Id<'id> {
    fn from(guard: Guard<'id>) -> Self {
        guard.0
    }
}

pub struct LifetimeBrand<'id>(PhantomData<&'id Id<'id>>);

impl<'id> LifetimeBrand<'id> {
    pub fn new(_: &'id Id<'id>) -> Self {
        LifetimeBrand(PhantomData)
    }
}

impl<'id> Drop for LifetimeBrand<'id> {
    fn drop(&mut self) {}
}
```

```
#[macro_export]
macro_rules! make_guard {
    ($name:ident) => {
        let branded_place: $crate::Id = std::marker::PhantomData;
        let lifetime_brand = $crate::LifetimeBrand::new(&branded_place);
        let $name = $crate::Guard(branded_place);
    };
}
```

Before we get started, let's get some low hanging fruit out of the way. We can verify that `min_generativity` is zero-cost: every single type is some form of `PhantomData`, a zero-sized type that is optimized away at compile-time. However, there is a sharp corner: we create a reference to a `PhantomData` in `make_guard`, and references to zero-sized types are perhaps surprisingly not zero-sized [27] due to some idiosyncrasies. Thus, we cannot prove `min_generativity` is zero-cost as Rust lacks a specification for optimization behavior. I claim that in practice this is the case the overwhelming majority of the time. The reference: is never used anywhere, is associated with an unused variable (`lifetime_brand`), and has no `Drop` impl. Even at the most basic optimization level, rustc is smart enough to no-op everything [28].

Note that `min_generativity` benevolently assumes that library users will only use `make_guard` to construct `Id<'id>` and `Guard<'id>`, as both are public types with public field visibility. The actual `generativity` crate privatizes `Id<'id>` (via a newtype) and `Guard<'id>` and marks their constructors unsafe, hidden within `make_guard`. Such was omitted to be concise.

5.2. The first part

```
use std::marker::PhantomData;

pub type Id<'id> = PhantomData<fn(&'id ()) -> &'id ()>;
```

If you've only used `PhantomData` when the compiler has told you to, this certainly looks nonsensical. The purpose of `Id<'id>` as we saw earlier is to

carry a *unique* lifetime brand among different `PermGroup<'id>`s, but aren't lifetimes already unique? What's the deal?

In Rust, *variance* determines whether you can substitute one lifetime for another. If you have a longer lifetime, Rust lets you use it where a shorter one is expected. This is also known as *subtyping*. Normally this is good—subtyping introduces static analysis that allows for more programs to compile—but in our case this automatic substitution works against our favor. If an `Id<'id>` can be tied to a lifetime other than its lifetime brand, we lose. So, the unique lifetime `'id` must have no subtyping relation with other lifetimes; `'id` is what's called an *invariant* lifetime.

The contemporary usage of the word "invariant" by Rustaceans has two meanings: one as a type-level guarantee (a noun), and one as a no-subtyping relation (an adjective). Invariant generally means something that cannot change or must be fixed to a specific value. Both meanings refer to this same general concept. We've been working with the first meaning so far, but for the rest of this article we'll switch to the second one.

To make `'id` invariant, we take advantage of a fundamental constraint with function pointer types. When you have `fn(&'id T) -> &'id T`, the caller provides a reference with lifetime `'id` and expects to get back a reference with that same lifetime `'id`. If Rust allowed the function pointer type to accept a longer lifetime but return a shorter one, or vice versa, it would break this explicit contract. You might pass in a reference that lives for ten seconds but get one back that lives for five seconds, creating a dangling pointer. Function pointer types with the same lifetime in the input and output positions force that lifetime to be invariant. No substitution allowed.

Of course, we don't actually want to store a function pointer at run-time. We only utilize it to make `'id` invariant at compile-time. The language provides `PhantomData` to enable fine-grained control over variance. In this case it tells the compiler to pretend like it holds a function pointer while not actually taking up any space.

Throughout the years lifetime invariance has been achieved in several other ways.

```
pub type Id<'id> = PhantomData<&'id mut &'id ()>; // Rust standard
library
pub type Id<'id> = PhantomData<*mut &'id ()>; // GhostCell paper
pub type Id<'id> = PhantomData<Cell<&'id u8>>; // Aria Desires' master's
thesis
pub type Id<'id> = PhantomData<Cell<&'id mut ()>>; // Also from her
master's thesis
```

They work because they follow the core principal that `&mut T` is invariant over `T` (click here [29] to see why). With `T = &'id ()`, `'id` must become invariant. Unlike the others, `PhantomData<fn(T) -> T>` implements all auto traits (`Send`, `Sync`, etc) for its owner, and it is generally preferred to convey that the only purpose is to indicate invariance. For further reading, the Rustonomicon provides a table [30] of common `PhantomData` patterns.

When I first learned how to use `PhantomData` to indicate variance I couldn't help but think of it as an obnoxiously leaky abstraction. There is a movement [31] to make it a bit more ergonomic by introducing custom variance newtypes into the standard library, i.e. `PhantomInvariantLifetime`. Sure enough, the status quo of `PhantomData` has been considered "something of a failure [32]."

5.3. The second part

```
pub struct Guard<'id>(pub Id<'id>);

impl<'id> From<Guard<'id>> for Id<'id> {
    fn from(guard: Guard<'id>) -> Self {
        guard.0
    }
}
```

The entire implementation of `Guard<'id>` is a newtype around `Id<'id>`. The established difference being that `Guard<'id>` doesn't implement `Copy` or `Clone`. We provide a `From` implementation to consume a `Guard<'id>` into an `Id<'id>` to create distributable copies of the lifetime brand, as we saw earlier.

5.4. The third part

```
pub struct LifetimeBrand<'id>(PhantomData<&'id Id<'id>>);

impl<'id> LifetimeBrand<'id> {
    pub fn new(_: &'id Id<'id>) -> Self {
        LifetimeBrand(PhantomData)
    }
}

impl<'id> Drop for LifetimeBrand<'id> {
    fn drop(&mut self) {}
}

#[macro_export]
macro_rules! make_guard {
    ($name:ident) => {
        let branded_place: $crate::Id = std::marker::PhantomData;
        let lifetime_brand = $crate::LifetimeBrand::new(&branded_place);
        let $name = $crate::Guard(branded_place);
    };
}
```

It turns out that disabling lifetime subtyping is not enough. While Rust believes it's unsound to freely resize `'id`, there's nothing that constrains where `'id` should come from. Consider the following system:

```
fn unify_lifetimes<'id>(_: &Id<'id>, _: &Id<'id>) {}

let id1: Id<'id1> = PhantomData;
let id2: Id<'id2> = PhantomData;
unify_lifetimes(&id1, &id2);
```

The constraint solver realizes there is no logical contradiction with the obvious solution of `'id2 = 'id1`, and it allows this program to compile. We need to uniquely tie `'id1` and `'id2` to their respective declaration sites to prevent Rust from unifying them.

After `make_guard` creates `branded_place` and generates an invariant lifetime `'id`, we are now armed with the knowledge required to examine how `LifetimeBrand` ensures it is non-unifiable. It takes the approach of establishing distinct lower and upper bounds for `'id`, highlighting the need for a macro with protected hygiene to prevent these bounds from potentially being manipulated.

Notice that `LifetimeBrand` carries the phantom type `&'id Id<'id>` (to avoid actually storing the reference). The existence of `LifetimeBrand`'s `Drop` impl means this borrowed data could potentially be used at the end of the scope, delegating special analysis called the drop check [33]. The drop check forces the compiler to extend `'id` to live at the point where `lifetime_brand` is dropped, constituting our lower bound. The actual `Drop` impl is purposefully left blank.

An important guarantee from the compiler is that local variables in a scope are dropped in the opposite order they are defined. Now we must prevent successive `make_guard` invocations in the same scope from unifying with the first invocation whose lifetime lives the longest. We are left with a need to *upper-bound* `'id`, and this is done by tying `'id` to the *borrow of what created it*. So, any expansion of `'id` would mean `branded_place`'s borrow of lifetime `'id` wouldn't live long enough when it is dropped.

If you still find it confusing, I encourage you to work out rustc's error message in this example [34]. I also encourage you to play around with this code snippet to see what compiles and what doesn't. I've found this exercise illuminating. Another tip to help you understand is to remember the pithy saying that lifetimes are descriptive, not prescriptive.

5.5. Verifying soundness

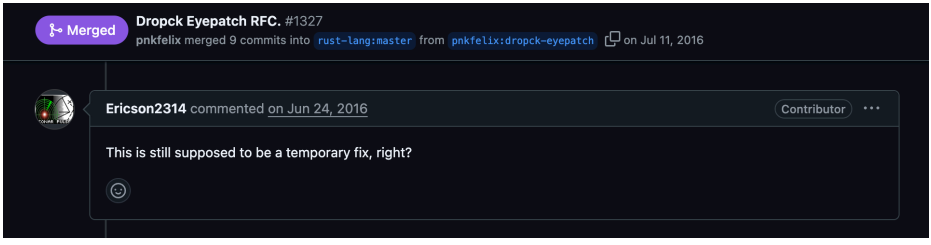
It is easy to verify that `generativity` is sound.

```
make_guard!(id1);
make_guard!(id2);
assert_eq!(id1, id1);
assert_eq!(id2, id2);
// rejected: `branded_place` does not live long enough
// assert_eq!(id1, id2);
```

At the time of writing, this test case passes. However, `generativity`'s vice is that it relies on internal behavior from the drop check analysis, the precise rules of which have historically been ill-defined and subject to change. In theory, sufficiently advanced analysis would be able to see that dropping `lifetime_brand` doesn't require `'id` to live because its `Drop` impl is empty, destroying the uniqueness guarantee we have created.

> The full extent of the current drop check analysis is detailed in a t-types meeting document [35]. TL;DR [33].

Pertaining to our concerns, any weakening of the drop check forcing captured lifetimes to live will most likely be opt-in, based on the direction of the "drop check eyepatch" RFC [36]. It introduces the unsafe `#[may_dangle]` attribute which relaxes this requirement. `#[may_dangle]` *opts-in* a struct's `Drop` impl to say, "I don't access a generic parameter, so it can be dropped before I run." Drop check eyepatch was introduced as a hacky refinement over "unguarded escape hatch [37]," which permitted high-priority collections like `Vec<&T>` to drop despite the `&T` borrow being invalidated beforehand.



Here are some famous last words. Drop check eyepatch was accepted nine years ago, yet stabilization is still opposed [38]. There are too many subtle gotchas with respect to the drop check that have been found to be too much of a footgun even for an internal compiler feature. It has resulted in unsoundness multiple [39] times [40]. The stable analysis is deliberately conservative for this reason. To quote from the Rustonomicon's explanation [41] of `#[may_dangle]`, "it is better to avoid adding the attribute."

The existing avenue of improvement [42] clarifies the semantics but still holds that this behavior will be *opt-in*. Hence, we can strengthen our confidence that the livelihood requirements for `generativity` will remain sound.

There are two other soundness concerns that are unlikely to be problematic but are still brought up in brief:

- `generativity` relies on an unused variable, `lifetime_brand`, to impact borrow checking and the drop check. If support for unused variable analysis is ever removed, then `scopeguard` [43], a crate with hundreds of millions of downloads, would break, and Rust is very careful not to break existing code. `scopeguard` also relies [44] on the `Drop` impl of an unused variable. Additionally, with the drop check, borrow checking must also run by virtue of `lifetime_brand`'s `Drop` impl, which could possibly access the borrow[5].
- In the ultra rare case where `generativity` is used in a divergent function, the drop checker will realize that `Drop` never runs and skip the drop check entirely. Special care [46] is required in this case to uphold soundness.
- It may still be hard to trust the delicate configuration of upper and lower bounding the generated lifetime. With non-lexical lifetimes [47] stabilized in 2022 as the second edition of the borrow checker, the only planned next iteration is Polonius [48], the implementation of which currently passes [49] the aforementioned test case. I haven't given it much thought, but

[5]Unreachable code on the other hand is *not* borrow checked because it simply wasn't a priority [45].

proving `generativity`'s soundness with Polonius' formal model of the borrow checker would be a fun project (to me at least).

5.6. Language support

We can no longer ignore the elephant in the room with the `make_guard` macro: it looks ugly. It injects local variables into the current scope, and we saw this was necessary in the statement-position to prevent the lifetime bounding tricks from being manipulated. For a while there was no resolution, until just a few months ago when the experimental `super let` [50] feature was introduced to extend the lifetimes of block-scoped variables. By creating a block scope, expression-position `make_guard` is made possible on nightly Rust.

```
#![feature(super_let)]

// ...

#[macro_export]
macro_rules! make_guard {
    ($name:ident) => {
        let branded_place: $crate::Id = std::marker::PhantomData;
        let lifetime_brand = $crate::LifetimeBrand::new(&branded_place);
        let $name = $crate::Guard(branded_place);
    };
    () => {{
        super let branded_place: $crate::Id = std::marker::PhantomData;
        super let lifetime_brand = $crate::LifetimeBrand::new(&branded_place);
        $crate::Guard(branded_place)
    }};
}

fn main() {
    make_guard!(guard);
    let guard = make_guard!();
}
```

Furthermore, there are preliminary ideas [51] that would allow `make_guard` to be a function instead of a macro. The feedback for `super let` has so far

been positive, so once the semantics are ironed out I think efforts to stabilize this feature will be underway.

My last contribution to this discussion is some wishful thinking about first-class language support for `generativity`. The troubles with unique lifetime branding stem from the fact that Rust offers no way to prevent lifetimes from unifying. So, I propose the `#[nonunifiable]` lifetime attribute. It would allow lifetimes to declaratively guarantee non-unifiability without having to resort to `generativity`'s lifetime bounding tricks. `#[nonunifiable]` is *not* intended to indicate variance—that's the job of `PhantomData`. For the permutation example, first-class language support from the compiler would look like this. The full implementation is here [52].

```
pub struct PermGroup<#[nonunifiable] 'id> {
    base_permutation_length: usize,
    base_permutations: Vec<Permutation<'id>>,
    id: PhantomData<fn(&'id ()) -> &'id ()>
}

pub struct Permutation<'id>(Box<[usize]>, PhantomData<fn(&'id ()) -> &'id
()>);
```

First-class language support is also the perfect excuse to improve the confusing [53] compiler errors:

```
let first = PermGroup::new(4, vec![vec![1, 2, 0, 3]]).unwrap();
let second = PermGroup::new(3, vec![vec![2, 0, 1]]).unwrap();
let first_perm = &first.base_permutations()[0];
let second_perm = &second.base_permutations()[0];

first_perm.compose(second_perm);
```

```
error[E0308]: mismatched types
 --> src/main.rs:9:23
  |
6 |     let first_perm = &first.base_permutations()[0];
  |         ---------- binding `first_perm` declared here with nonunifiable
  lifetime `'1`
```

```
7 |     let second_perm = &second.base_permutations()[0];
  |         ----------- binding `second_perm` declared here with nonunifiable
  lifetime `'2`
8 |
9 |     first_perm.compose(second_perm);
  |                ------- ^^^^^^^^^^^ expected `Permutation<'1>`, found a
  different `Permutation<'2>`
  |                |
  |                arguments to this method are incorrect
  |
  = note: expected reference `&Permutation<'1>`
             found reference `&Permutation<'2>`
  = note: `Permtuation<'1>` and `Permutation<'2>` look like similar types, but
  are distinct because they carry `#[nonunifiable]` lifetimes
```

Jack (one of this article's peer reviewers) and I discussed what first-class language support to remove the lifetime parameter and allow `Guard` to escape its scope could look like. Unfortunately, we came to the conclusion that such a system would be equivalent to the problem case described in Section 4.2. Creating an arbitrary number of branded types in a loop during run-time would require deep changes to the type system.

Maybe `#[nonunifiable]` will have an unexpected use case that would make it practical, or maybe not. I'm not going to pretend like I've figured out all of the semantics. The point is to get my thoughts up in the air.

6. Benchmarks

No comparison would be complete without a benchmark. Yes, the point of the generativity pattern is more fundamental than just speed, but I know what people want. I statically generated two random length-fifteen permutations and wrote a Criterion benchmark for all five approaches to permutation composition.

Benchmark Name	Time (ns)
1-slice	14.805
2-newtype	4.257

4-atomic_id	3.940
5-generativity	3.604
3-unsafe_trait	3.602

Empirically, this validates all of my observations. The naive `1-slice` is the slowest because it checks every permutation for complete validity during composition. `2-newtype` removes most of the validation overhead. Admittedly this is good enough; again, from a practical standpoint, you would only care about the other solutions if you could prove that permutation composition was the bottleneck. `4-atomic_id` replaces the validation with a single integer comparison, making it marginally faster, likely because it avoids dereferencing. Finally, `5-generativity` and `3-unsafe_trait` emerge the fastest because they avoid validation entirely, and I have also verified that the generated machine code is identical. The important difference: `3-unsafe_trait` marks permutation composition unsafe while `5-generativity` does not.

7. Conclusion

Truthfully I don't have many final thoughts. I just needed a transition to end this article. I suppose my primary conclusion is that this article has gotten *far* longer than I had originally planned 😀.

I don't think this is a bad thing; its comprehensiveness more than makes up for it. The hidden agenda was to survey design patterns and write about Rust code I thought were interesting, culminating with the generativity pattern, which shows us how to take advantage of the type checker's power in a non-obvious manner.

This concludes my SIGHORSE submission! I came into this topic with surface level understandings of what `generativity`, `PhantomData`, and the drop check are and how they work. I was not expecting this to take five weeks of meticulous research and writing. I was entirely unprepared for how interesting the full story would be.

8. References

[1] [Online]. Available: https://cliffle.com/blog/rust-typestate/

[2] [Online]. Available: https://plv.mpi-sws.org/rustbelt/ghostcell/

[3] [Online]. Available: https://github.com/kyren/gc-arena/?tab=readme-ov-file#prior-art

[4] [Online]. Available: https://github.com/Manishearth/rust-gc

[5] [Online]. Available: https://cad97.com/

[6] [Online]. Available: https://crates.io/crates/generativity

[7] [Online]. Available: https://en.wikipedia.org/wiki/Permutation_group#Composition_of_permutations%E2%80%93the_group_product

[8] [Online]. Available: https://rust-unofficial.github.io/patterns/patterns/behavioural/newtype.html

[9] [Online]. Available: https://en.wikipedia.org/wiki/Permutation_group

[10] [Online]. Available: https://github.com/ArhanChaudhary/generativity-pattern-rs

[11] [Online]. Available: https://users.rust-lang.org/t/should-i-use-unsafe-merely-to-encourage-users-to-maintain-invariants/27856

[12] [Online]. Available: https://github.com/ogxd/gxhash/issues/82#issuecomment-2257578785

[13] [Online]. Available: https://github.com/ArhanChaudhary/generativity-pattern-rs/blob/main/src/5-generativity.rs

[14] [Online]. Available: https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=4675f6eb33925940c51668ee15a00010

[15] [Online]. Available: https://doi.org/10.1007/BF01018827

[16] [Online]. Available: https://github.com/Gankra/thesis/blob/master/thesis.pdf

[17] [Online]. Available: https://gitlab.mpi-sws.org/FP/ghostcell/-/blob/master/ghostcell/examples/branded_vec.rs

[18] [Online]. Available: https://en.wikipedia.org/wiki/Parametric_polymorphism#Higher-rank_polymorphism

[19] [Online]. Available: https://doc.rust-lang.org/nomicon/hrtb.html

[20] [Online]. Available: https://github.com/ArhanChaudhary/generativity-pattern-rs/commit/806c8bef89b1d0c0621db42c130856bf33fffb9f

[21] [Online]. Available: https://stackoverflow.com/users/865874/rodrigo

[22] [Online]. Available: https://stackoverflow.com/a/76876800

[23] [Online]. Available: https://github.com/ArhanChaudhary/generativity-pattern-rs/blob/main/src/6-unsound_token.rs

[24] [Online]. Available: https://codeberg.org/binarycat/typetoken/src/branch/trunk/src/lib.rs

[25] [Online]. Available: https://codeberg.org/binarycat

[26] [Online]. Available: https://crates.io/crates/typetoken

[27] [Online]. Available: https://github.com/rust-lang/rfcs/pull/2040#issuecomment-317275303

[28] [Online]. Available: https://godbolt.org/z/4h4xccfjT

[29] [Online]. Available: https://doc.rust-lang.org/nomicon/subtyping.html#variance

[30] [Online]. Available: https://doc.rust-lang.org/nomicon/phantom-data.html#table-of-phantomdata-patterns

[31] [Online]. Available: https://github.com/rust-lang/rust/issues/135806

[32] [Online]. Available: https://github.com/rust-lang/rfcs/pull/3417#pullrequestreview-1396551771

[33] [Online]. Available: https://doc.rust-lang.org/std/ops/trait.Drop.html#drop-check

[34] [Online]. Available: https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=649d51907c2612c310eb627a0c863399

[35] [Online]. Available: https://hackmd.io/h9YBnIbaRSCD7Ej6hUpF_w

[36] [Online]. Available: https://rust-lang.github.io/rfcs/1327-dropck-param-eyepatch.html

[37] [Online]. Available: https://rust-lang.github.io/rfcs/1238-nonparametric-dropck.html#unguarded-escape-hatch

[38] [Online]. Available: https://rust-lang.zulipchat.com/#narrow/stream/144729-t-types/topic/Perma-unstable.20status.20of.20.60.23.5Bmay_dangle.5D.60

[39] [Online]. Available: https://github.com/rust-lang/rust/issues/76367

[40] [Online]. Available: https://github.com/rust-lang/rust/issues/99408

[41] [Online]. Available: https://doc.rust-lang.org/nomicon/dropck.html#an-escape-hatch

[42] [Online]. Available: https://github.com/rust-lang/rfcs/pull/3417

[43] [Online]. Available: https://crates.io/crates/scopeguard

[44] [Online]. Available: https://docs.rs/scopeguard/latest/src/scopeguard/lib.rs.html#287

[45] [Online]. Available: https://github.com/rust-lang/rust/issues/91377#issuecomment-993875185

[46] [Online]. Available: https://github.com/CAD97/generativity/pull/16

[47] [Online]. Available: https://blog.rust-lang.org/2022/08/05/nll-by-default/

[48] [Online]. Available: https://github.com/rust-lang/polonius

[49] [Online]. Available: https://rust.godbolt.org/z/vhMjKGbz3

[50] [Online]. Available: https://github.com/rust-lang/rust/pull/139080

[51] [Online]. Available: https://blog.m-ou.se/super-let#a-potential-extension

[52] [Online]. Available: https://github.com/ArhanChaudhary/generativity-pattern-rs/blob/main/src/7-nonunifiable_proposal.rs

[53] [Online]. Available: https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=47b36de838eaeeebe236e2f3b4aa279b

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/

Acknowledgments

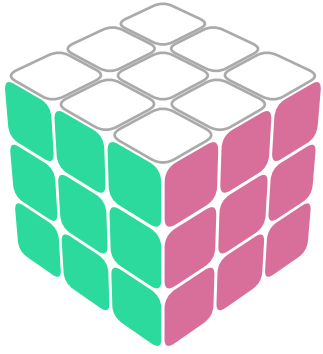
We extend our sincere thanks to Tomas Rokicki for personally providing us key insight into Rubik's Cube programming techniques throughout the past year. Qter would not have been possible without his guidance. We are immensely grateful for his time.

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

Contents

1) Introduction

The Rubik's Cube.



We've all seen it before; it is one of the most recognizable objects on Planet Earth. But do you know how to solve one? If you're the average person, you

probably don't, but it's actually much easier than you think. Instructions for how to solve one can fit into just two pages [1]—that's only 4% of the length of this article! But what if I told you that "solving" was only scratching the surface of things that you can do with a Rubik's Cube. It's like painting on a canvas with only white paint: you can make endless varieties of strokes and swirls, but it always has the same result: a blank canvas—a solved cube. It turns out that there's a whole world of color out there, and we are ready to show it to you.

What if I gave you a different set of Rubik's Cube instructions, not for *solving* it, but perhaps for something else. You don't need to know how to read this, for we will teach you later…

```
1 | input "Which Fibonacci number to calculate:"
            B2 U2 L F' R B L2 D2 B R' F L
            max-input 8
2 | solved-goto UFR 14
3 | D L' F L2 B L' F' L B' D' L'
4 | L' F' R B' D2 L2 B' R' F L' U2 B2
5 | solved-goto UFR 15
6 | repeat until DL DFL solved
            L U' B R' L B' L' U'
            L U R2 B R2 D2 R2 D'
7 | L' F' R B' D2 L2 B' R' F L' U2 B2
8 | solved-goto UFR 16
9 | repeat until FR DRF solved
            D' B' U2 B D' F' D L' D2
            F' R' D2 F2 R F2 R2 U' R'
10 | L' F' R B' D2 L2 B' R' F L' U2 B2
11 | solved-goto UFR 17
12 | repeat until UF solved
            B R2 D' R B D F2 U2 D'
            F' L2 F D2 F B2 D' L' U'
13 | goto 4
14 | halt "The number is: 0"
15 | halt until DL DFL solved
            "The number is"
            L D B L' F L B' L2 F' L D'
```

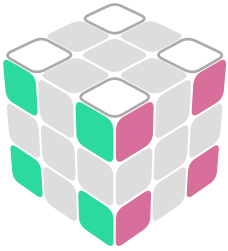
```
16 | halt until FR DRF solved
            "The number is"
            F2 L2 U2 D' R U' B L' B L' U'
17 | halt until UF solved
            "The number is"
            U L' R' F' U' F' L' F2 L U R
```

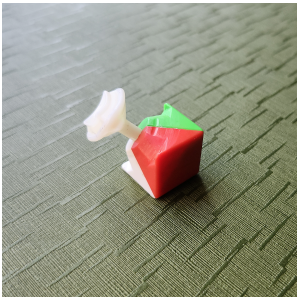
...but if you repeat the "`input`" scramble n times, follow the instructions from top to bottom, and reach the "`halt`" instruction, your Rubik's Cube will *not* be solved, but rather hold a very special scramble. If you repeat the `halt` scramble on it over and over again, the cube will actually become solved. How many times do you have to repeat it until it becomes solved? The nth Fibonacci number times. You just used your Rubik's Cube as a computer. But how is that even possible?

1.1) Background

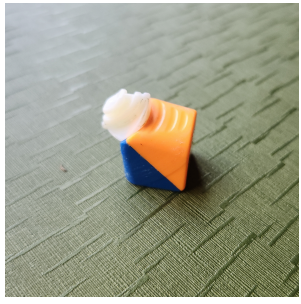
Before we can explain how to turn a Rubik's Cube into a computer, we have to explain what a Rubik's Cube *is* and the fundamental mathematics behind how it works. First, a Rubik's Cube is made out of three kinds of pieces: *Corners*, *Edges*, and *Centers*.

Corners

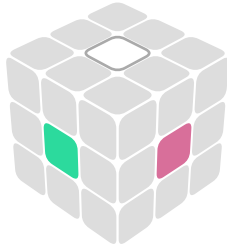


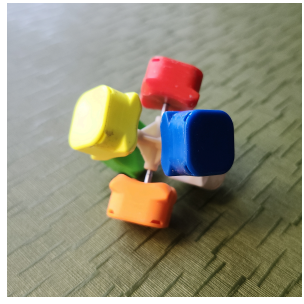


Edges

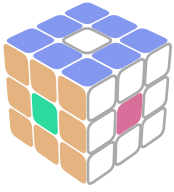


Centers





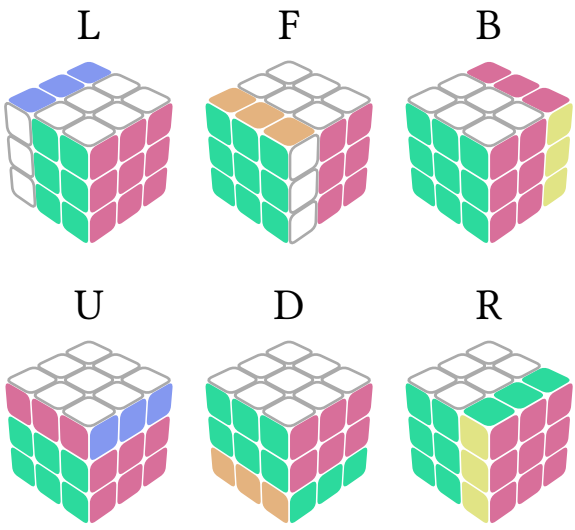
You can see that the centers are attached to each other by the *core* and are only able to rotate in place. This allows us to treat the centers as a fixed reference frame to tell whether or not a sticker is on the correct side. For example, if we have the following scramble,



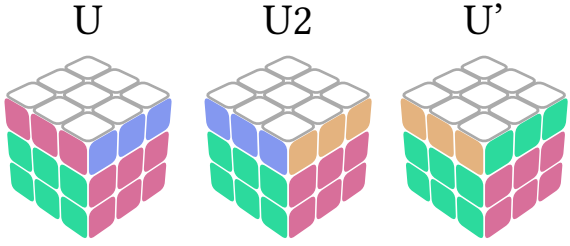
it may look as if the centers are the only thing unsolved, but in fact we would actually consider *everything else* to be unsolved. The reason is that all of the stickers are different from the center on the same side as it. Next, people who are beginners at solving Rubik's Cubes often make the mistake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

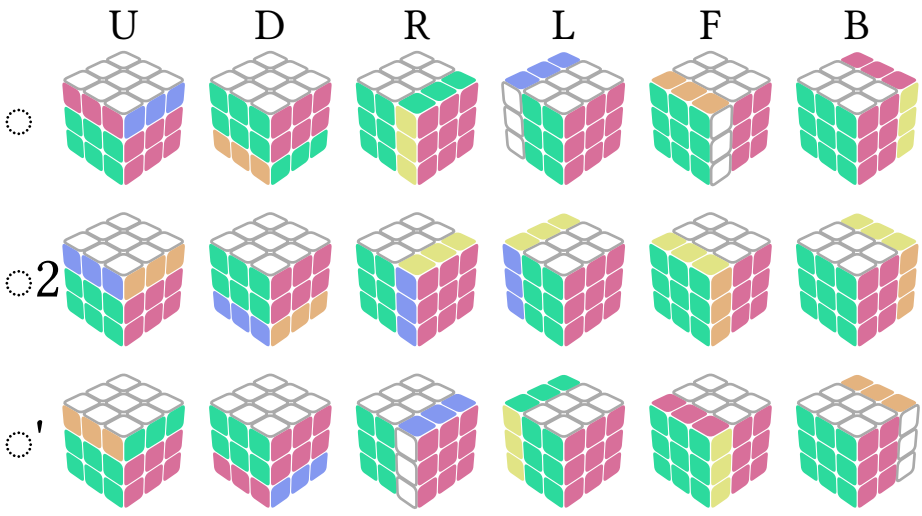
Now, we need some way to notate scrambles and solutions on a Rubik's Cube. We will use the conventional "Singmaster Notation" which is standard in the Rubik's Cube solving community [2]. First, we will name the six sides of a Rubik's Cube *Up* (U), *Down* (D), *Right* (R), *Left* (L), *Front* (F), and *Back* (B). Then, we will let the letter representing each face represent a clockwise turn about that face.



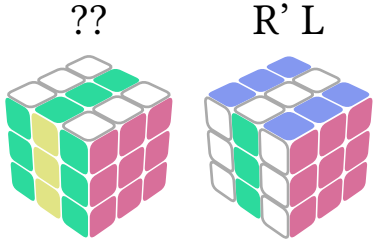
To represent double turns or counterclockwise turns, we append a `2` or a `'` respectively to the letter representing the face.



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



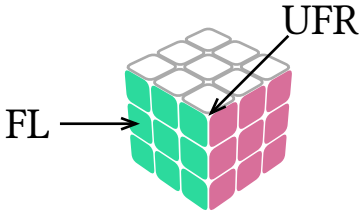
It may look like we're forgetting some moves. After all, there are *three* layers that you can turn, not just two, and we haven't given names to turns of the three middle slices. However, we don't actually need to consider them because "slice" turns can be written in terms of the 18 "face" turns.



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.

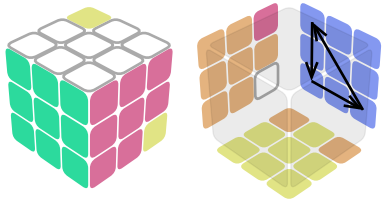
Another thing that we will need to name are the pieces of a Rubik's Cube. To do this, we can simply list the sides that the piece has stickers on. For example,

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:

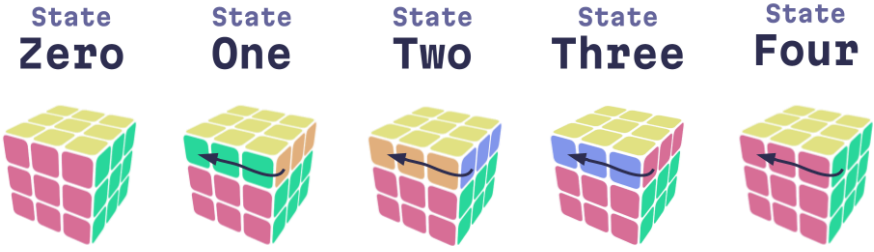
L2 D2 L' U' L D2 L' U L'



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*.

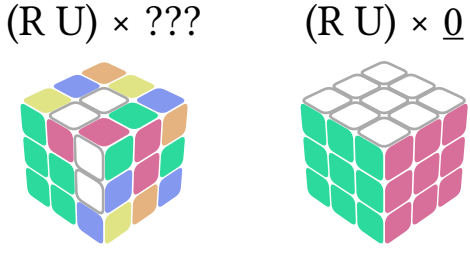
There are still lots of cube states left; can we do better? Unfortunately, it's only possible to get to 1259, wrapping around on the 1260th iteration. You can

try this using the algorithm `R U2 D' B D'`. This has been proven to be the maximum order possible [3].

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.



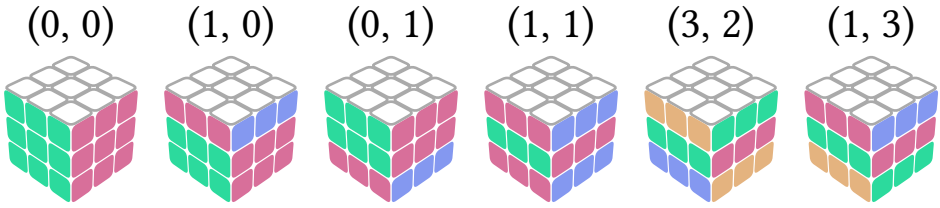
2.0.4) Multiple numbers

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.

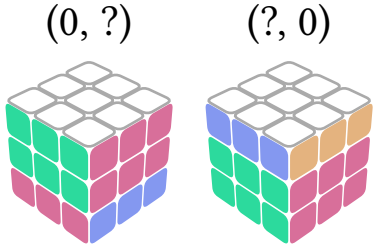
Something cool about Rubik's Cubes is that it's possible for a long sequence of moves to only affect a small part of the cube. For example, we showed in

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.

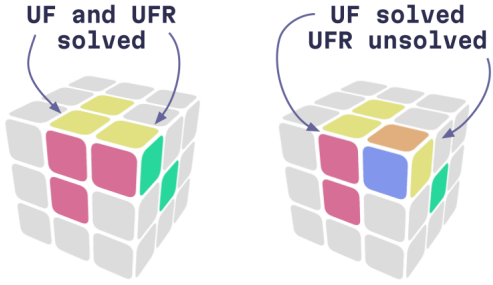


As described, `solved-goto` would only branch if the entire cube is solved, however since each algorithm affects a distinct area of the cube, it's possible for a human to determine whether a *single* register is zero, by inspecting whether a particular section of the cube is solved. Remember that "solved"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



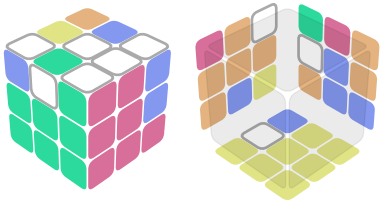
For the first cube in the above figure, it's easy to tell that the first register is zero because the entire top layer of the cube is solved. We can modify the "solved-goto" instruction to input a list of pieces, all of which must be solved for the branch to be taken, but not necessarily any more. The 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.

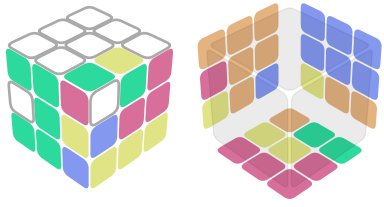


Can we do better than two registers with four states? In fact we can! If you try out the algorithms `R' F' L U' L U L F U' R` and `U F R' D' R2 F R' U' D`, you can can see that they affect different pieces and both have order ninety. You may notice that they both twist the DBL corner; this is not a problem 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.

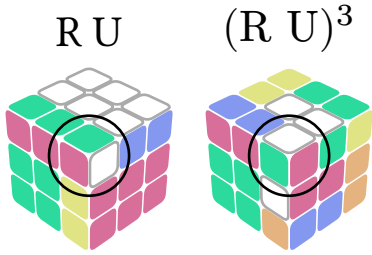
R' F' L U' L U L F U' R (1, 0)



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.



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:

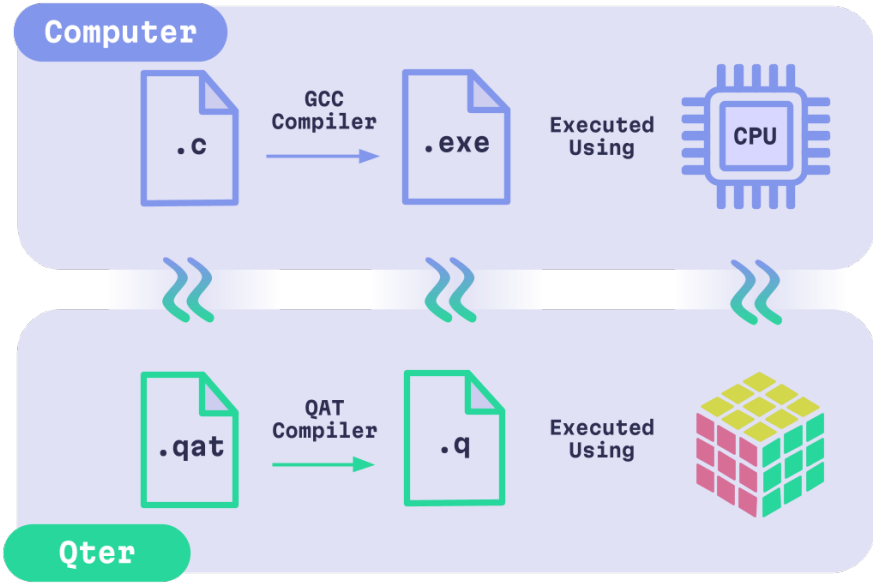
```
Puzzles
A: 3x3
B: 3x3

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...

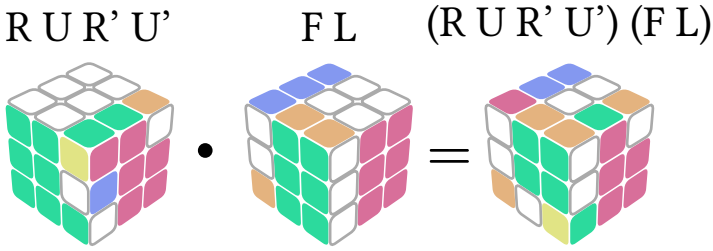
3.1.1) Group theory

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*:

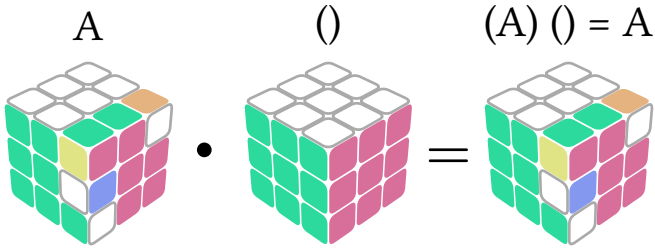
- There exists an *identity* element e such that for any element of the group a, $a \cdot e = a$.
- For all elements a, b, c, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. In other words, the operation is *associative*.
- For each a in the group, there exists a^{-1} such that $a \cdot a^{-1} = e$. In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

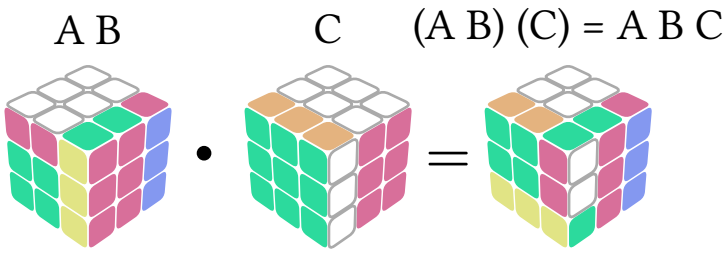
Next, we need an *operation*. For the Rubik's Cube, this will be jamming together the algorithms that reach the two cube states. We will call this operation *composition* because it is very similar to function composition.

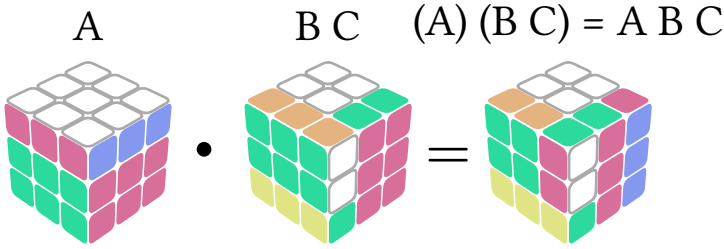


Now, let's verify that all of the group axioms hold. First, we need an identity element. This identity is simply the solved state! Lets verify this, and let A be an arbitrary scramble:



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.

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

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:

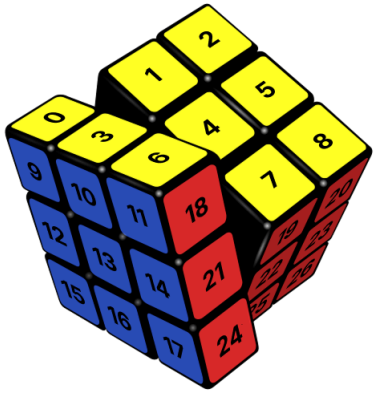
$$a^{-1} = \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 2 & 1 & 4 & 3 & 0 \end{array} \rightarrow \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 4 & 1 & 0 & 3 & 2 \end{array}$$

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 [4]:

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

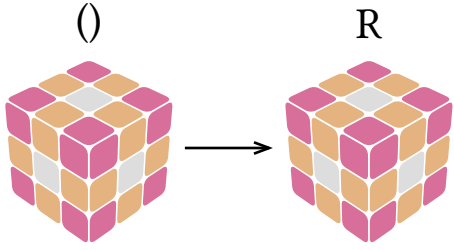
The exact numbers aren't actually relevant for understanding, but you can sanity-check that exponentiating all of them to the fourth gives identity, due to all of the cycles having length four. This matches our expectation of how Rubik's Cube moves should work.

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

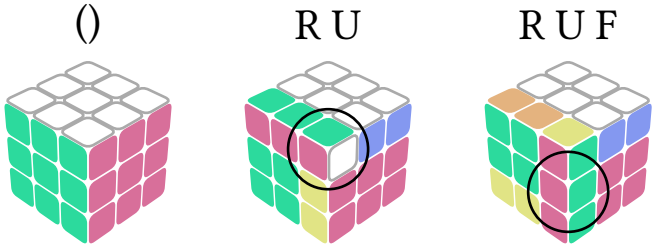
sticker's place, then that other sticker must be in the same orbit as the first. On a Rubik's Cube, there are two orbits: the corners and the edges. There obviously doesn't exist an algorithm that can move a corner sticker to an edge sticker's place or vice versa, therefore the corners and edges form 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

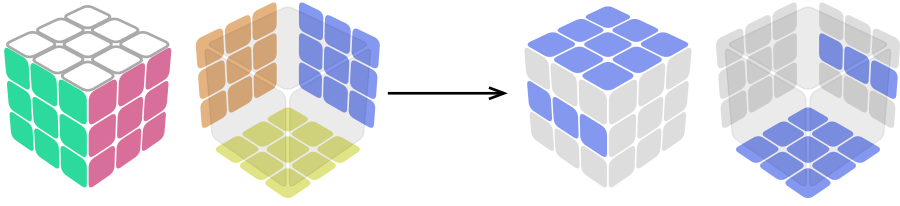
Now, we need to show some properties of how the Rubik's Cube group works. First, we would ideally like a way to take pieces into account in our representation of the Rubik's Cube group. After all, we showed in the introduction how important they are to the mechanics of the cube. What we could do is instead 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.

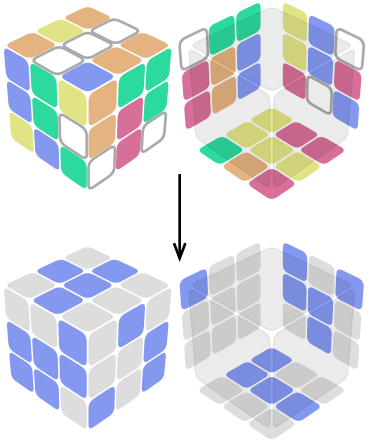
In general, any edge or corner can exist in any other edge or corner position in any orientation. So how can we encode this orientation in full generality? It's easy to tell that the UFR corner and FR edge are twisted and flipped respectively in the above examples because the pieces can be solved by 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

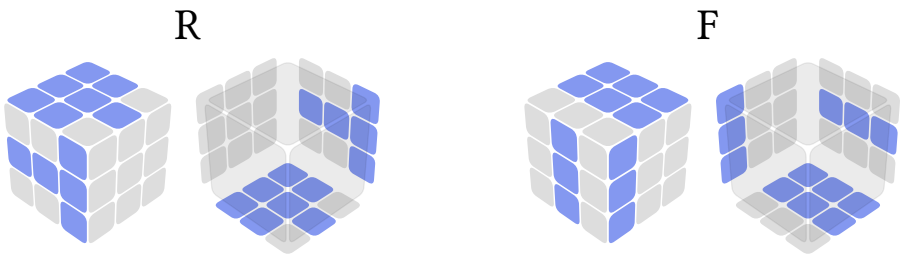
this is that we already know how to represent the locations of pieces using a permutation group, so it is valid to throw out the knowledge of a piece's location while figuring out how to represent orientation. To determine the orientation of a piece on a normally colored Rubik's Cube, you can take the algorithm to get to that cube state and apply it to our specially recolored cube:



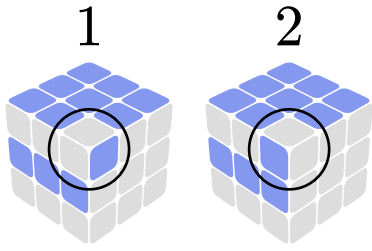
Even though the UFR corner isn't in its solved position, we can still say that the piece in the UFR position is twisted because the blue sticker isn't facing up, like it is in the recolored solved state. You would be able to "solve" that piece —make it look like the respective position in the recolored 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.

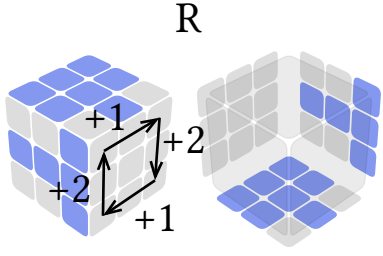
Based on this recoloring, you can see that the move set $\langle U, D, R2, F2, L2, B2 \rangle$ preserves orientation of all of the pieces, and on top of that, R and L preserve orientation of the edges but not of the corners. The moves F and B flip four edges, while R, F, L, and B twist four corners.



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.

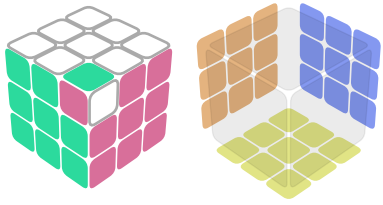


This should make logical sense. We already know that if you apply R twice, the corners don't get twisted, and that can be seen in the figure as well. If you

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:

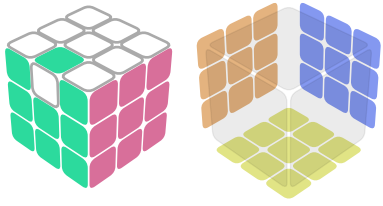
$$\emptyset$$



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:

$$\emptyset$$



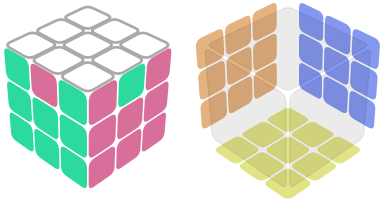
Is there anything else that's unsolvable? Actually, yes! For this to make sense, we have to think of permutations as a composition of various swaps. For example, a four-cycle can be composed out of three swaps:

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

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.

$\emptyset$

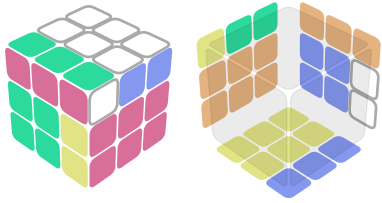


Is there any other arrangement of pieces that is unsolvable? Actually no! You can show this by counting the number of ways that you can take apart and randomly put together a Rubik's Cube, then dividing that by three because two thirds of those positions will be unsolvable due to the corner orientation 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

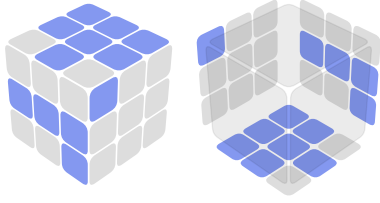


Next, lets trace where the pieces go. Instead of using numbers to represent the pieces in the cycle notation, we can simply use their names.

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

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



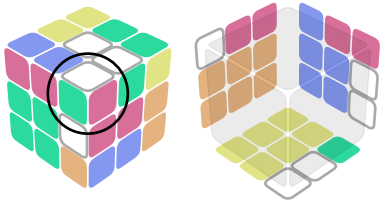
I'm going to notate orientation by writing the amount of orientation that a piece acquires above it.

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

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.

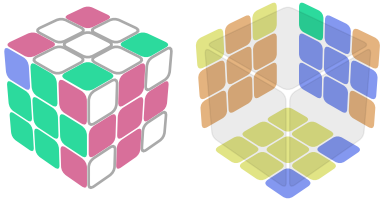
First, we have to consider how many iterations it takes for each cycle to return to solved. To find this, we have to consider both the length of the cycle and the overall orientation accrued by each piece over the length of the cycle. Lets consider the first cycle first. It has length one, meaning the 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.

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



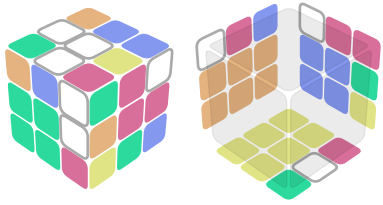
Next, let's consider the cycle of edges. They have a cycle of seven and don't accrue orientation at all, so it simply takes 7 iterations for the edges to return to solved.

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



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.

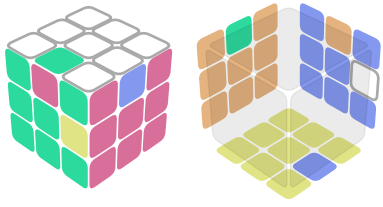
$$(\mathrm{R\ U})^5$$



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

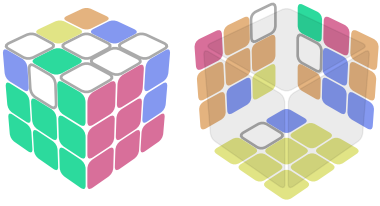
```
Puzzles
A: 3x3

1 | solved-goto FDR FR 1
```

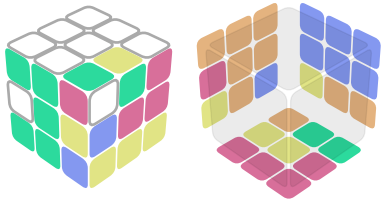
3.1.5) Orientation and parity sharing

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:

$$A = \underset{+2}{(\text{DBL})}\underset{+1}{(\text{UF})}\underset{+1}{(\text{UFL, UBL, UBR})}\underset{+0}{(\text{UL, LB, RB, UB, LD})}\quad\begin{matrix}+0 & +1 & +0 & +0 & +0\end{matrix}$$

$$B = \underset{+1}{(\text{DBL})}\underset{+1}{(\text{UFR})}\underset{+1}{(\text{DFR, DFL, DBR})}\underset{+2}{(\text{RD})}\underset{+1}{(\text{UR, FL, DB, FR, FD})}\quad\begin{matrix}+0 & +0 & +0 & +1 & +0\end{matrix}$$

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 [5] 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.

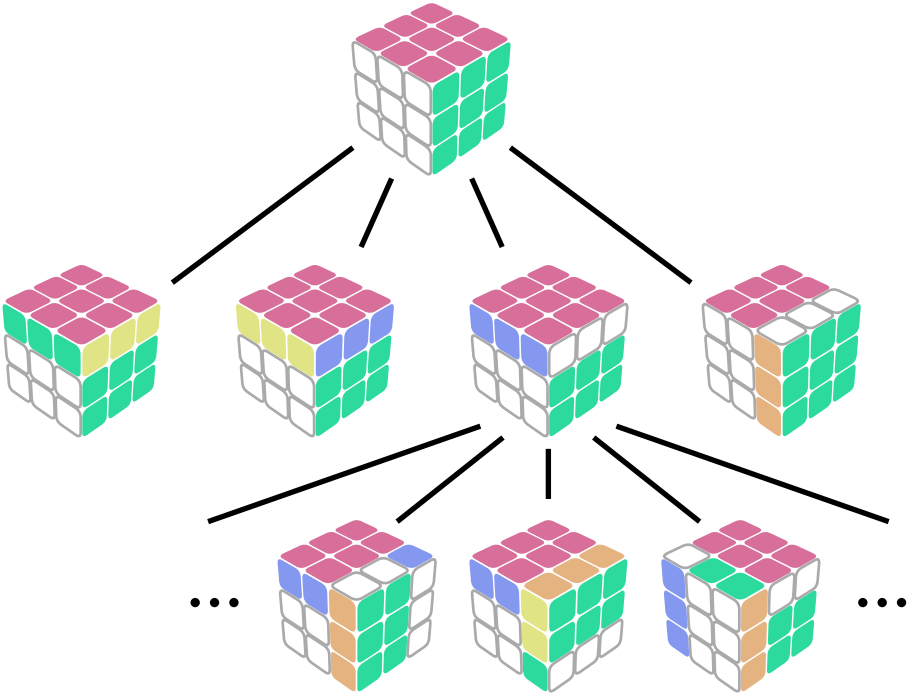
In an optimal Rubik's Cube solver, we are given a random position, and we must find the shortest algorithm that brings the Rubik's Cube to the solved state. Analogously, the Cycle Combination Solver starts from the solved state and finds the shortest algorithm that brings the puzzle to a position with 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 [6].

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 [7] cleverly utilizes a specific subgroup to solve up to 3900 individual position per second *near* optimally [8]. 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 [9] 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.

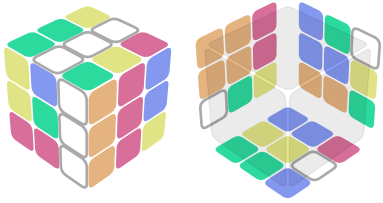
A simple modification to DFS can make it always find the optimal solution. We tweak the DFS implementation so that it explores up until a specified depth, testing whether each node at this depth is a solution, without exploring further. We repeatedly run this implementation at increasing depth limits 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 node.

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 [6] 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 [10]. 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 of IDA*. In general pruning table distributions are nontrivial to analyze, so our observations below are not a formal analysis but rather a series of intuitive arguments. 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 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. If we first assume the branching factor b to be constant, implying each depth has exactly b times more states stored in the pruning table than the previous depth, we notice the maximum depth that is stored in the pruning table is at least $\log_b m$. Since there are exponentially more states at the last depth, p is negligibly less than $\log_b m$; hence, $p \simeq \log_b m$. In reality, there are two flaws with this assumption. First, the branching factor is not constant and always less than its theoretical value, eventually converging to zero. This implies our estimate of $p \simeq \log_b m$ is an egregious overestimate of the actual average pruning value, but we consider this okay because IDA* is biased in favor of smaller heuristic values. Second, when there are relatively many Cycle Combination Solver solutions, the maximum depth state stored in the pruning table decreases. We also consider this okay because many solutions implies that one will be found at a lesser search depth. If we let λ equal to both of these reductions, we find that the IDA* search depth limit remains approximately the same: $(d - \lambda) - (p - \lambda) = d - p$. All of the aforementioned biases cancel each other out to some extent, so we proceed with this approximation of p.

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

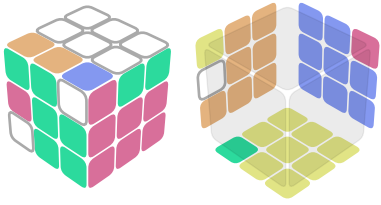
The larger the admissible heuristic, the better the pruning, and the lesser the search depth. So, we need to carefully design our pruning tables to maximize:

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

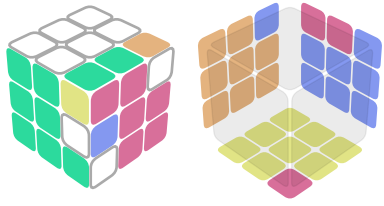
3.3.4.a) Symmetry reduction

Symmetry reduction is the most famous way to compress pruning table entries. We thank Kociemba [11] 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'

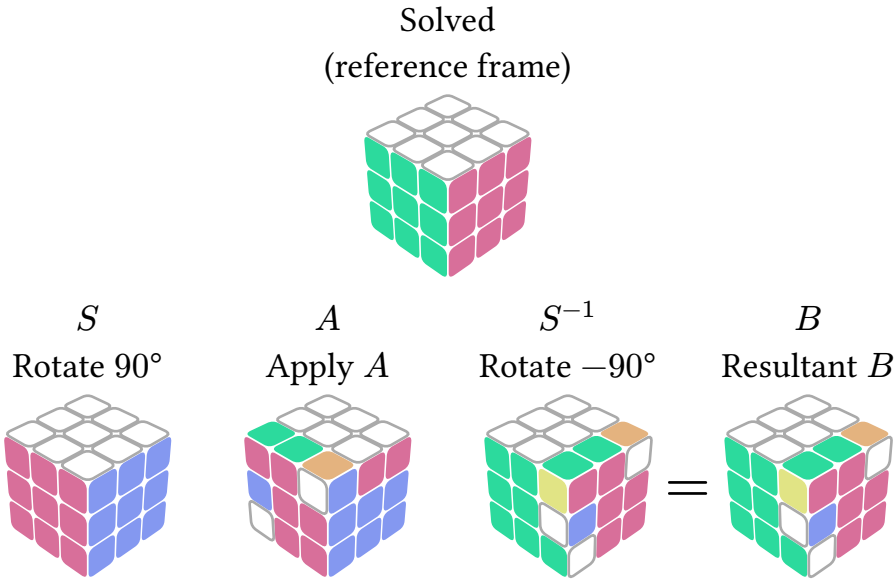


They are different but they are *basically* identical. If you replace red with blue, blue with orange, orange with green, green with red, you will have transformed A into B. Because these two cube positions have the exact same structure of pieces, they need the same number of moves to reach a Cycle Combination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

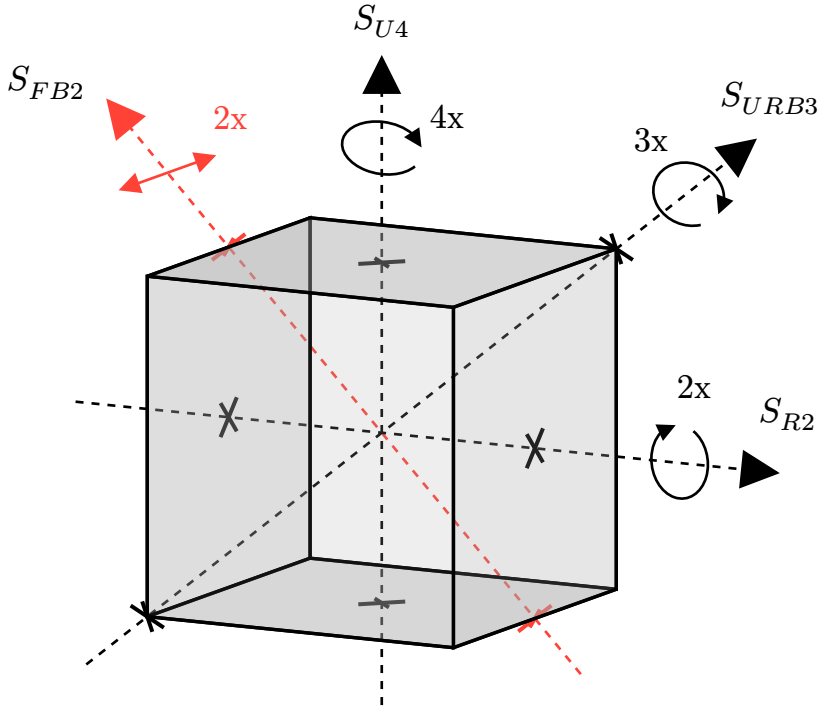
very efficient. The more mathematically precise way to define symmetrically equivalent cubes is with permutations. Two cube positions A and B are symmetrically equivalent if there exists a symmetry S of the cube such that $SAS^{-1} = B$, where the S operations are spatial manipulations the whole cube. 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.

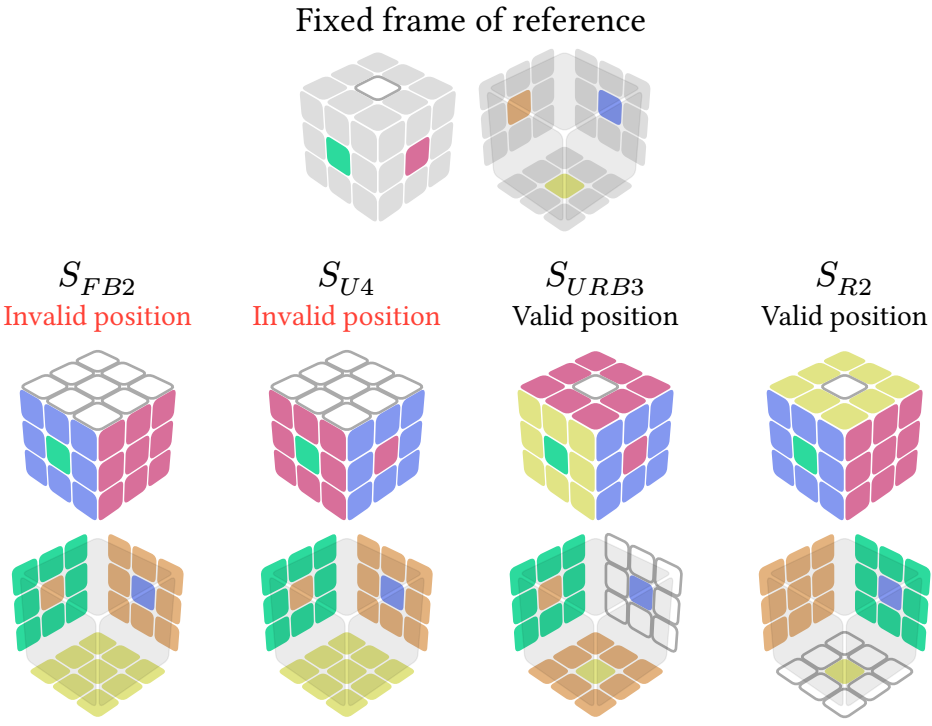


The 48 symmetries of the cube

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

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 [12]: 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 [13] and test for a Cycle Combination Solver solution [14]. 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:

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

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 [15] 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 $\Omega\left(\frac{13.348^d}{md}\right)$ and $O\left(\frac{13.348^d}{m}\right)$. 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 $AB \ldots C$ such that A and C commute for a solution. The proof is as follows.

We first observe that if $AB \ldots C$ is a solution, then $CAB \ldots$ 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 $DB \ldots$. Such a shorter solution would have been found at the previous depth limit, implying that $AB \ldots C$ 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 $CAB \ldots$ would be a lexicographically lesser than $AB \ldots C$, 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.

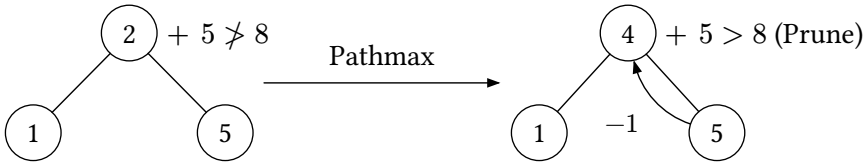
If CAB ... is a solution, then ACB ... is also a solution because A and C commute. By the transitive property, if AB ... C is a solution, then so is ACB ... . Both of these sequences are independently searched and tested as a solution because there is no direct "commutative move ordering" or sequence 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 [16] 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* [17]). 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 [18]

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 [18]

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 [19] 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 [20] 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 [12]), 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 [21], which took around 35 CPU-years [8]. 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 [22]. 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 [23] 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 [24] 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.

5) Appendix 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
```

6) References

[1] Lucas Garron, “Rubik's Cube Solution — Reference Sheet.” [Online]. Available: https://cube.garron.us/solution.pdf

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

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

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

[19] 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

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

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

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

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

[24] 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/