Programming Optimization - Operation Compression

By Joseph Juma

Description

We can use the principles of simple compression to optimize our programming workflow. The model I've considered is similar to Huffman Compression. In Huffman compression we do the following.

1. Read the file, identifying sequences that repeat. Let's call these `patterns`
2. Calculate how long a pattern is (`pattern length`), and how frequently it appears (`pattern frequency`).
3. Multiply pattern length and frequency together to get `pattern weight`
4. List all patterns in a table, sorted from the highest weight to the lowest weight.
5. Substitute a new string (`symbol`) for each pattern, making sure that the highest weight patterns have the smallest symbols substituting for them.
6. Make sure to never substitute a symbol that is longer than the pattern it is replacing.

This is a very simple version, and it also is what's called "single-pass". Single-pass means that you only run through these steps once. However, sometimes new patterns appear after the first-pass that can be used to compress things down even further, which is called "multi-pass". We won't talk about multi-pass optimization, but keep in mind it's just the same algorithm as "single-pass" applied to a system that was already optimized. Even though we won't be describing it, you *can* apply multi-pass to programming optimization.

Now, why is programming able to be compressed the same way?

1. Programming is a sequence of actions over time. If we assign a symbol to these actions, whether broad like `create class`, or specific like `create integer named x`, then we get a sequence of symbols over time - a one dimensional sequence of symbols - which is exactly what a string is.
2. Therefore, we can identify actions that happen repeatedly when programming, and these are `programming patterns`.
3. We then measure how long a pattern takes, as we are optimizing over time, not over the number of steps a pattern takes. This is a `programming pattern length` or just `ppl`
4. We then measure how frequently a pattern is performed, this is `programming pattern frequency` or just `ppf`
5. We then multiply these together to get `programming pattern weight` or `ppw`
6. We then list these sorted from highest weight to lowest weight.
7. Once sorted, you should now try to reduce the amount of time spent on the highest-weight items first, then move towards the lowest weight items.
8. Reducing time often means writing automated tools that do the programming pattern for you.
9. Just keep doing this kind of analysis constantly, and you will optimize your programming.

Mathematics

So, now that I've described all that in english I think some people, like myself, would like to see the math. The math helps you make decisions better in specific circumstances, so let's see.

1. A `programming pattern length` or `ppl` is the time in seconds a behavior takes, we'll denoted this with the variable α.
2. A `programming pattern frequency` or `ppf` is the number of times a behavior happens within some time period. You should perform these analysis on days, weeks, months, quarters, biannually and years. We'll use `weekly` for this example and denote the `ppf` with the variable β.

3. You can then calculate the `programming pattern weight` or `ppw` which is denoted with the variable γ using the following equation.

$$\gamma = \alpha\beta$$

Pretty simple so far, right? However, what we really want to measure is how good an optimization might be, and its impact on the overall process. So, to do that we need to make an estimation on how long the process will take once we optimize it, which will be measured in seconds (just like `ppl`) and be denoted with the variable ϵ. I'll use an example going forward. In this example let's say that I'm creating new classes often.

1. so I'm often writing up the basics of a class file a lot, say, 10 times a week, so `ppf` is 10. So $\beta = 10$.
2. It takes me on average 2 minutes, or 120 seconds to write a new basic class file, so `ppl` is 120. So $\alpha = 120s$.
3. Therefore the `ppw` is, $\gamma = \alpha\beta = 10 * 120 = 1200$
4. Let's estimate that we can optimize this down to the time to type out a class name into a script, say 10 seconds. So $\epsilon = 10s$

Now, how do we calculate how effective this would be?

1. First, we need a variable representing the total amount of time you measured programming for, which we'll denote ζ. Since we're measuring a week in this example, let's say you put in 20 hours of programming a week which is 1200 minutes or 72000 seconds, so $\zeta = 20 * 60 * 60s = 72000s$.
2. We then take the amount of time that a pattern takes up out of your week, which is what the `ppw` represents, and subtract it from the total time to get the remaining time, which will be denoted with η. The equation for this is,

$$\eta = \zeta - \gamma$$

So…

$$given\ \zeta = 72000s$$
$$given\ \gamma = 1200s$$
$$\eta = \zeta - \gamma = 72000s - 1200s = 70800s$$

3. We then need to calculate the new weight, which will be denoted θ, of the optimized approach. This is calculated the same way as normal `ppw`. Since we're replacing the original programming pattern, the frequency is the same as the original `ppf` but using the weight of the optimized version. So the equation is,

$$\theta = \epsilon\beta$$

So…

$$given\ \epsilon = 10s$$
$$given\ \beta = 10$$
$$\theta = \epsilon\beta = 10s * 10 = 100s$$

4. Then add this back to the time value (η) to get the adjusted time spent programming in the week, denoted with Δ.

$$\Delta = \theta + \eta$$

So…

$$given\ \theta = 100s$$
$$given\ \eta = 70800$$
$$\Delta = \theta + \eta = 100s + 70800s = 70900s$$

So now we have the new programming time. So, how much time was saved? To answer this I will proceed to change around the algebra equations without further written explanation; please follow the equations.

$$given\ \theta = \epsilon\beta$$
$$given\ \eta = \zeta - \gamma$$
$$\Delta = \epsilon\beta + \zeta - \gamma$$

$$given\ \gamma = \alpha\beta$$
$$\Delta = \epsilon\beta + \zeta - \alpha\beta = \zeta + \beta(\epsilon - \alpha)$$

This gives us the adjusted time, now to find the change in time, which will be denoted δ_t =, we take the difference between these two. Since we know the adjusted time *should*

be smaller, we can just use simple subtract, making sure Δ is the second variable.

$$\delta_t = \zeta - \Delta = \zeta - \zeta + \beta(\epsilon - \alpha) = \beta(\epsilon - \alpha)$$

Now, in English all this means is you can calculate the change in time by subtracting the original time (`ppl`) from the new time (which will give a negative value, because time has *shrunken*) and multiply that by the `ppf`, which is rather sensible and simple! With the example values, this equates out to..

$$\begin{gathered} given\ \alpha = 120s \\ given\ \beta = 10 \\ given\ \epsilon = 10s \\ \delta_t = \beta(\epsilon - \alpha) = 10(10s - 120s) = 10(-110s) = -1100s \end{gathered}$$

So we've saved 1100 seconds or 18 minutes and 20 seconds from our week of programming. Over one year this is equal to 57,200 seconds (1100 seconds per-week times 52 weeks), which is 953 minutes and 20 seconds, or 15 hours 52 minutes and 48 seconds.

Two other useful metrics are to know the percentage change in overall time, denoted Δ_p and the percentage of change between the two approaches, denoted δ_p. The δ_p can be thought of as being used in the sentence, "the optimized approach takes δ_p percent of the original time." Similarly, the Δ_p can be thought of as being used in the sentence, "with the optimization, we can get the same amount of work done in Δ_p percent of the time." The following formulas can be used to calculate these two values.

$$\delta_p = \frac{\beta\epsilon}{\beta\alpha}$$

If you instead wanted to say the sentence, "the optimized approach is $\delta_p^{'}$ percent faster than the original", you would use the equation,

$$\delta_p^{'} = 1 - \delta_p = 1 - \frac{\beta\epsilon}{\beta\alpha}$$

and, for the values of Δ_p,

$$\Delta_p = \frac{\Delta}{\zeta}$$

and if you wanted to say, “the optimized approach has made our week $\Delta_p^{'}$ percent faster” you would use the equation,

$$\Delta_p^{'} = 1 - \Delta_p = 1 - \frac{\Delta}{\zeta}$$

So, given the example values…

$$\delta_p = \frac{100s}{1200s} = 0.08333...333 = 8.33\%$$
$$\delta_p^{'} = 1 - 0.08333...333 = 0.91666...666 = 91.67\%$$

$$\Delta_p = \frac{70900s}{72000s} = 0.9847222...2222 = 98.47\%$$
$$\Delta_p^{'} = 1 - \Delta_p = 1 - 0.9847222...2222 = 0.0152777...777 = 1.53\%$$

So the optimizations can then be said to have the following impacts:

1. With the optimized approach we can get the same work done in 98.47% of the time, and it’s made our week overall 1.53% faster.
2. The optimized approach is 91.67% faster, taking only 8.33% of the time as the original approach in each use.

Caveats

Now, you may look at those numbers and say they’re tiny - which they are - but the fact of the matter is that most optimizations you will be doing compound. If 2% here, 1% there, and there’s 30+ areas you can change, then it compounds out to a *large percentage of your total programming*. I will also be clear: in the interest of being neutral, I chose values for the example which were underneath those I’ve personally seen. In one instance, I measured that writing a new base-file for a `struct` in C++, due to the standards of the code, took around 3 minutes per struct due to creating two files (a header and body). Writing a generator made this take 6 seconds, so that’s a change of 180s to 6s, and so that’s a base increase that is larger than the example’s 120s to

10s. This is why actually measuring and calculating things out is beneficial in decision making with things like this.

However, this technique shouldn't be applied everywhere it can be. Specifically, programmers as people often times have a knack for bloating the time it takes to do simple things by trying to automate them. A pretty funny programmer-YouTuber, Michael Reeves, once said a great line that can be paraphrased as, "This could have taken maybe 4-5 hours of mindless work, but I made it take only a month, because I'm a programmer, and that's what we do" - and this is completely accurate. If you just optimized the way I've described without actually planning things out, then you're going to often wind up applying this approach incorrectly and actually causing your programming process to be overall slower. So, how do you figure out when to use this?

1. You need to calculate how much time you think you will save. When measuring this at small time-frames (day, week, month) it's usually better to use your percentage of time saved, as otherwise you will get values back of only *minutes* saved for a few hours of work and that never pans out. I usually opt for instead measuring things on an annual time-frame, so that I know how much time is saved over a year by doing the optimization now. In the example I gave above this was just shy (by less than 8 minutes) of 16 hours, so 16 hours will be saved. Since we're doing a bit more math, let's use the variable a (not α) for that.
2. Then you need to estimate how long you think it will take you to write the optimization tool. Let's use the variable b not β for that.
3. If $a > b$ then it's probably a good idea to write the optimization, otherwise don't - you'll just waste time.
4. However, programming is hard, and even more hard to estimate accurately, so it's advised you actually multiply your estimate of how long you think it will take. I usually multiply all my estimates by 2. So, instead when determining if optimizing is worthwhile I use the logic: if $a > 2b$ then it is a good idea to write the optimization.

So, going back to the example where we save 16 hours over a year, it is only a good idea to write that optimization if I estimate that it will take under 8 hours to do so. Now, what happens if you go over-budget? Well, that's fine too overall, right? If you reach 16 hours and are around 80% done, so it takes 20 hours to complete, then that's fine. 4 hours isn't that much in the grand scheme of things, especially for those 16 hours you

still saved. However you should probably note that down and adjust how much you multiply your estimates by in the future so they're more accurate.

Not doing this simple estimation is honestly where I've seen the most automation efforts go wrong, and where my own automation efforts have personally gone wrong in the past. So just keep it in mind. Now, one last thing: if you are working on a team, then you should perform these estimations with a being replaced with A which is the sum of a for each of your team-mates. If you want to be precise, you need to do the large equations for estimation per team-mate, which is often a bit…much, and it's rare you're tracking what kind of operations everyone is doing all the time. If you *are* tracking and optimizing that, congrats - you're like the Olympic Team of programmers. However it can be safe to just do a simplified version where A is just a multiplied by how many team-mates you have. So on a team of 4, that's $A = 4a$. In this case you're now saving 64 hours per year, and you can then have one person work on it for up to 32 hours - and if it only takes 20 hours then that's fantastic!

Conclusion

So, in review:

1. Track how long each part of programming takes you (α), and how frequently you're doing it (β).
2. Estimate how much you can optimize it to make it shorter (ϵ).
3. Calculate what percentage of the original time it takes and project that out over a year, $\delta_p = \frac{\beta\epsilon}{\beta\alpha}$ to get the estimate of time saved (a)
4. Estimate how long it will take for you to write the optimization (b)
5. Count how many team-members will be using it (n)
6. Then pad your estimate for safety (by 2 if you're like me)!
7. $if\ na > 2b \rightarrow$ optimize, else don't optimize.

If you follow those steps, you should be able to optimize down your programming practices and make you, and your team-mates lives better! Hopefully this has been helpful.

Thanks and have a great day.
