•••

Technical Details

Technical details of `defg`.

v1.7.5

charles.lobo@gmail.com

https://github.com/theproductiveprogrammer/defg#readme

Discussion

We have three fundamental problems we are solving in `defg`:

1. Reconciling differences between the old README and updated documentation comments.
2. Preserving images, formatting, and `insert-block`s in the README.
3. Assembling the scattered documentation comments together in the right way to match the README.

To solve (1) we will use the standard Myers-Diff algorithm This has been around a long while and probably has the best balance between speed and getting a reasonable result. However, notice that in order to solve (1) effectively, we need to have a solution for (3) and (2) as well.

Let us define D_x as a "docblock" - a block of documentation comments in a file. If we have $(D_0, D_1, \ldots, D_m)$ docblocks let us define $\Theta_m!$ as the set of all permutations of these docblocks. Taking R as the README, problem (3) then reduces to finding:

$$d_i \text{ where } \min_{\forall d_i \in \Theta_m!} (R - d_i)$$

To solve (2), once we have found the best (minimum) d_i, we add lines that insert images, formatting, and inserted blocks to a set of "special" lines Sp. Then we find the corresponding docblock for each special line in Sp and copy the line across in roughly the same position.

Performance Discussion

There are two major costs we can identify from the above:

1. Calculating $\forall d_i \in \Theta_{m!}$, and
2. Calculating $(R - d_i)$

Calculating d_i

Calculating $\forall d_i \in \Theta_{m!}$ involves finding all the permutations of the elements in $\Theta_{m!}$. This gives us an upper bound of $O(m!)$. Now this is one of the worst upper bounds we could possibly have and that means that the algorithm is extremely sensitive to the performance of $(R - d_i)$.

Calculating $(R - d_i)$

The standard method for calculating $(R - d_i)$ is to use the Levenshtein distance or Dice's Coefficient. However these have roughly exponential $O(n^2)$ where n is the length of the README.

Bad News

The total performance of a generic implementation would therefore be $O(m!n^2)$. This is such a slow process it would almost certainly be useless as the size of the README and number of "docblocks" increases.

We shall see how to handle this issue next.

Algorithm Design

We therefore have to design an algorithm which can handle all our requirements but also has a fighting chance of working with reasonable performance. This is how we do it:

1. Construct a permutation d_i by selecting each docblock $(D_0, D_1, \dots, D_m)$ in turn, and recursively permuting the rest.
2. During the construction, at every step we calculate $(R' - d'_i)$ - the difference that we can calculate thus far.
3. During the recursion, we use $(R' - d'_i)$ as the starting difference and calculate only the difference from that point onward.
4. If, at any point, we discover that the difference is greater than an already calculated difference, we can "short-circuit" that entire branch because we know that the minimum value cannot be down that path.

To calculate the partial difference $(R' - d'_i)$, we use a line-wise algorithm:

1. Start with pointers at the beginning of R' and d'_i.
2. Compare the lines at the current pointers, advancing both if they match
3. If they do not match, look ahead some Δ lines to see if there is a match.
4. If there is, it must be an insert (or delete) and so advance the corresponding pointer and count the number of lines inserted/deleted into the difference.
5. If there is no match, then count the lines as changed, and increase the difference by 2 (counting both lines).

This algorithm has the advantage that (a) it can work with partial permutations and resume them as we needed, (b) it can incorporate all Sp lines by simply recognizing them during the and, (c) it can exit very early in the cycle thus trimming the tree effectively.

In brief, this is the algorithm we are using in `defg`. You can see the code and details in `regen.js`.

Futher Improvements

Because of the nature of our problem, we can make a reasonable assumption that most blocks have very little in similar. If we assume this to the case, then we can search for an expected solution in $O(m^2)$ instead of $O(m!)$ using the following strategy:

1. Try each block one by one on the first slot and use the best one (the one with the smallest distance).
2. Repeat with the remaining blocks for the next slot and so on.

This greedy approach will very likely give us the optimal solution (given little similarity between blocks). If we then feed this as the input into the first algorithm we should be able to confirm we don't have a better solution in roughly $O(n)$ time.

A final (minor) improvement, is to short-circuit all searches as soon as we find a 0-difference (nothing can be better).

Time Trials

For comparison, I ran an experiment with 70 docblocks for each of the algorithmic approaches:

- Naive/Generic Approach: Unknown (stopped run after 2+ days)
- Algorithm 1: 16 hours 10 minutes
- With Greedy Initialization: 7 seconds!

Because the greedy optimization is so much better in performance, `defg` allows you to pick a "quick mode" where it will only use the greedy algorithm. This uses the assumption that most blocks are substantially different. You can use this mode if it turns out that the full search is taking too long.