Disintegrating Dinosaurs, but fast

Algorithm Engineering 2022 Project Paper

Maurice Wenig
Friedrich Schiller University Jena
Germany
maurice.wenig@uni-jena.de

Nick Würflein
Friedrich Schiller University Jena
Germany
nick.wuerflein@uni-jena.de

ABSTRACT

Visualizing data is a very important part of data analysis. The authors of "Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing" [2] supports that point with a Python script that is able to transform datasets into a variety of distinct graphs that have the same statistical properties. However, the amount of available graphs is very limited and the Python implementation is very slow. We provide a C++ implementation that is 3000 times faster than the original Python implementation, with the ability to easily add new target graphs.

KEYWORDS

performance, statistics, dinosaurs

1 INTRODUCTION

1.1 Background

Anscome's Quartet [1] is a set of four distinct datasets, that have the same summary statistics (mean, standard deviation, correlation). It is often used to point out that it is important to visualize data instead of just looking at the statistical properties. "Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing" [2] reinforces that point. The paper provides an algorithm that is able to transform datasets into other datasets, that have the same statistical properties, but very different looking graphs. The authors of that paper also provided a Python script that implements this algorithm.

1.1.1 Original Imlementation. The Python implementation of the original algorithm looks like this:

```
current_ds ← initial_ds
for x iterations do
    new_ds ← Perturb(current_ds, temp)
    if error is still ok then
        current_ds ← new_ds
function Perturb(ds, temp)
    loop
        i ← random index
        old_point ← ds[i]
        new_point ← Shake(old_point)
        if Dist(new_point) < Dist(old_point)
         or temp > Random() then
            ds[i] ← new_point
            return i
```


It is based on the idea that bringing each point of the dataset closer to a target form, eventually results in the whole graph of the dataset approaching the target form. The Perturb function tries moving random points in random directions until the distance from the moved point to the target form is improved. To not get stuck in locally-optimal solutions, movements that do not improve the distance are also sometimes accepted.

This original Python implementation is very slow. It needs over three minutes to compute 100000 iterations, which is just enough iterations to come close to the target form.

Also notice that in order to come closer to a target form, we need to define a distance measure, that evaluates, how far any given point is from the desired target form. In the original Python Implementation, this was done with a selection of formulas, that define forms that can easily be described mathematically. This has limited expandability, because the process of describing target forms mathematically can be very hard.

1.2 Our Contributions

The main goal of this project was to provide an implementation of the algorithm described in [2] with much better performance than the original Python implementation. To achieve this, we implemented the algorithm in C++ and tweaked some details to optimize run time. The detailed changes are described in subsection 2.2. With this in mind, the speedup we achieved is very significant and we talk about the experiments showing these improvements in subsection 3.3.

To make the process of designing custom target forms easier, we implemented a way to input 16×16 pixel wide black and white images as target forms. The details of this method are described in subsection 2.1 and in subsection 3.2 we provide examples.

2 METHODS

2.1 Distance to Images

In order for the points to come closer to the 16×16 pixel image, we defined the distance between a point and the image to be the distance between the point and the closest white pixel in the image. Pixels are then mapped to points in 2D-space, so that the distance between a point and a pixel is the distance between two points in 2D-space, which can easily be calculated. Because the length and positions of the pixels are not always the way we want them to be, we added the ability to scale and move the image. This can be adjusted so that the image fits the data.

2.2 Performance Improvements

2.2.1 Changing Points. In the original algorithm, the whole dataset is copied everytime a point is perturbed. To avoid this, we keep

two copies of the changing dataset. This allows us to copy only one point at a time. The new algorithm is listed below:

```
current_ds ← initial_ds
new_ds ← initial_ds
for x iterations do
    changed_index ← Perturb(new_ds, temp)
    if error is still ok then
        current_ds[changed_index] ← new_ds[changed_index]
    else
        new_ds[changed_index] ← current_ds[changed_index]
function Perturb(ds, temp)
    loop
        i ← random index
        old_point ← ds[i]
        new_point ← Shake(old_point)
        if Dist(new_point) < Dist(old_point)
         or temp > Random() then
            ds[i] ← new_point
            return i
```

2.2.2 Caching. Every iteration, the changed dataset is compared to the initial dataset to see if the statistical properties are still acceptable. To do this, both the statistical properties of the intial dataset and the changed dataset are computed. But the initial statistical properties do not change so they do not have to be calculated every iteration. Instead, they are calculated once before the first iteration starts.

The statistical properties depend on each other. The standard deviation depends on the mean, and the correlation depends on both the means and the standard deviations of both variables. Therefore, a lot of computation can be avoided by caching the needed statistical properties.

To determine if perturb is done, the distance from the old point and the new point to the target form are compared. To avoid calculating the distance from the old point to the form multiple times, the distance of each point to the dataset can be cached. This effectively halves the amount of distance calculations.

Distance calculations still have to be done multiple times every iteration. To calculate the distance between a point and an image, the distance from the point to the closest white pixel in the image has to be calculated. This can be done by iterating over the whole image and calculating the distance between point and pixel if the pixel is white. But this includes a lot of iterations (depending on the image) where the pixel is black and nothing is done. To avoid these iterations, the coordinates of the white pixels can be cached in an array. We can now iterate over the array of white pixels instead of the whole image.

2.2.3 Structure of Arrays. The statistical properties are calculated seperately on the x-axis and the y-axis. To do this efficiently, a Structure of Arrays can be used to guarantee stride-1 access. The only time we have to iterate over the 2 axes at the same time, is when we initialise things or calculate the correlation. Because of this, in every iteration, we would need $4 \times$ SoA and $1 \times$ AoS. But the use of AoS barely changes the performance of calculating the correlation (test results not shown here). So the use of both SoA and AoS at the same time is not worth it.

2.2.4 Vectorization. Because the statistical properties are calculated in every iteration, it is important to make these calculations fast. An effective way of doing that is by vectorizing the calculations. We vectorized the calculation of the statistical properties with simple omp reduction pragmas.

In case the target form is an image, the perturb function still takes a long time because the distance calculation iterates over all the white pixels to find the closest one. This can be sped up by vectorizing it as well. To do this, we use a omp reduction pragma and minimise the squared distance from the point to each of the pixels. This is easier than minimising the distance because we use the squared distance to calculate the distance with `sqrt`. But `sqrt` does not change the closest pixel because it is a monotonic function, so we work with the squared distances and apply the square root at the end to return the correct distance.

2.2.5 RNG. After a first performance analysis, we found out that generating random numbers is our biggest bottleneck. So instead of using the standard C++ RNG, we used an implementation of the xoroshiro RNG and the Box-Muller transform to generate gaussian random numbers.

3 RESULTS

3.1 Examples based on mathematical descriptions

The following figure shows one example run of our implementation. In this case the dinosaur dataset was given as an input and we calculated 200.000 iterations. The target structure, here a circle, is described mathematically to calculate the distance of a data point to said structure. The results are very similar to those shown in [2].

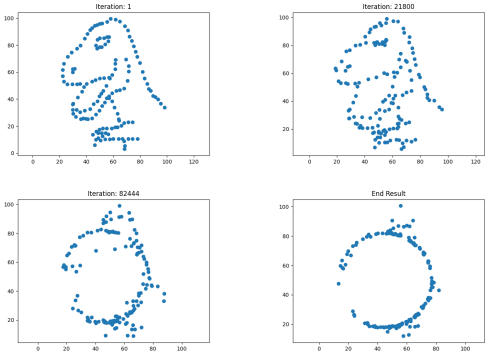

Figure 1: Transformation to a Circle

3.2 Examples based on images

Figure 2 shows another example, but this time we used a picture to describe the desired outcome.

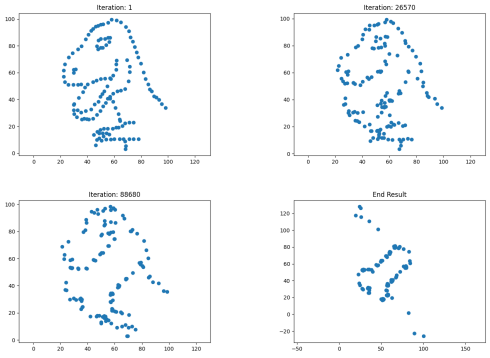

Figure 2: Transformation to an Image

3.3 Performance

We measured how our optimised tool performs against the original Python tool under different numbers of iterations. The measured run times are listed in Table 1. The system we used for running these tests was a Windows 11 machine with 16 GB of RAM and a Intel i7-7700HQ processor.

Table 1: Run Time Comparison

Iterations	Run Time		
	Python	C++	C++ images
10000	1m6.32s	0.03s	0.03s
50000	2m34.53s	0.04s	0.05s
100000	3m56.54s	0.08s	0.08s
200000	7m9.281s	0.13s	0.14s
1000000	not measured	0.52s	0.60s
5000000	not measured	2.58s	2.93s

The measured run times result in an average number of iterations per seconds recorded in Table 2

Looking at these results it is clear how impactful our improvements were overall. In the instances in which we calculated more than 200.000 iterations the run time for the original Python implementation exceeded our preset maximum run time of 15 minutes. This shows that the speedup is not only practical for saving run time, but it also extends the number of iterations that we are able to calculate in a practical timeframe.

Table 2: Iterations per Second Comparison

Iterations	avg. Iterations per Second		
	Python	C++	C++ images
10000	151	333K	333K
50000	324	1.25M	1M
100000	423	1.25M	1.25M
200000	455	1.54M	1.43M
1000000	not measured	1.92M	1.67M
5000000	not measured	1.94M	1.71M

4 DISCUSSION

During the course of this project we also had a few ideas to tweak performance that did not make it into the final version. We will not discuss every single one in detail, but we want to talk about some of them to illustrate possible future enhancements or to explain why some seemingly obvious improvements are not as promising as they seem at first glance.

The first thing that comes to mind when trying to optimize code on modern day architectures is parallelism. While there are some areas in this project that might actually benefit from a parallel approach, more on that later, the most obvious ones, namely the calculation of the statistical measures and the distances between points, certainly are not. In both cases the calculations are actually very easily parallelized, both are calculations that can be split up, distributed and solved on different threads without the need for any synchronization (except in the reduction step to gather the end result). But for the example data we looked at, which is the same data used in [2], the main problem in this approach lies in the number of input data elements. What we noticed when we implemented parallel versions of these operations is that since there are only about 160 data points the overhead needed for creating the threads, distributing the data and collecting the results is actually much higher than any performance gain we could achieve with this approach. This does not mean that a parallel approach to these operations isn't viable at all, only that it is not viable for such small sample sizes. If at any future point it is necessary to increase the number of input data points way beyond those in our examples (maybe into the range where a few million operations have to be performed) parallelizing the aforementioned operations would probably be valuable.

Another point where parallelism is possible is in the loop that goes through the iterations. This is usually a very large loop (in terms of iterations) and therefore seems even more interesting. The perturb function called inside that loop may change the position of the point that was chosen. If this procedure is parallelized many of these points may be changed at the same time. Afterwards we have to check if the statistical error caused by this procedure is low enough and herein lies the problem. First, we would have to synchronize these changed points at every iteration (or at least once perturb was successful) which comes at the cost of performance, otherwise the statistics could not be measured accurately. And second, it is unclear to us how the error will behave when a lot

of data points are perturbed at once. The error might exceed our limit in all instances. Additionally, we would have to define what happens if the error is exceeded. Are all the points going to be discarded in that case or do we revert some of the changes until the error is acceptable again? This would lead to a loss of some probably viable new points or to a lot of synchronization work and new error testing in each step as well. Overall we deemed these changes to the underlying algorithm too big and the outcome not to be certain enough. It might be interesting to test and analyse this approach since this might enable parallelism at a stage of the algorithm that would certainly benefit from it if it can be done efficiently.

One last idea we want to discuss involves the random choice of points before they are perturbed. To anyone who knows modern cache hierarchies this already sounds like a bad idea. If variables are read and written at random this also means that cache lines have to be loaded at random and it follows that there is next to no temporal or spatial locality (except maybe by random chance). A workaround to force some sort of locality would be to choose cache lines (two to be precise because we use a SoA, but the idea stays the same) instead of variables at random and to perturb every single one of the points in that cache line until choosing the next one. This approach comes at the obvious cost of decreasing randomness (which might not be impactful at all), but there are some other problems as well. First, again the size of the input problem is probably low enough so that all the variables could be stored in cache simultaneously (160 data points x 2 coordinates x 8 bytes $\approx$ 2.5KB), so again bigger problem sizes might benefit from this approach. Second, there are a lot of calculations executed right after the data point is perturbed and they are using a lot of data (think of the statistical measures for example). The data points would just be "flushed out" by the data that is required to be in the cache for the statistical calculations (this also only applies for bigger data sets, if the entire set fits into cache no data is evicted). So our idea of increasing the locality to better use our cache seems plausible at first, but taking a closer look it doesn't seem viable.

?? shows the running times of the resolution step of the five best placed teams.

5 CONCLUSIONS

To conclude this work we want to sum up our results and point out possible future work. While we have achieved an enormous speedup compared to the previous Python implementation of this algorithm overall, we only looked at a very small set of data points. It is plausible that the performance may drop off on larger sets due to the limitations described in section 4, but if this is the case we also suggested possible improvements to mitigate this impact.

Additionally we realized the possibility to generate data sets with matching statistics based on the input of a sample image. This feature works and it is also efficient as it has been shown in the examples in section 3.

Lastly we want to point out that further improvements, e.g. by finding a smart way to parallelize the loop in which the perturb function resides, are certainly possible, and we invite any reader to find them and to help us make dinosaurs disintegrate even faster.

REFERENCES

[1] F. J. Anscombe. 1973. Graphs in Statistical Analysis. *The American Statistician* 27, 1 (feb 1973), 17–21. https://doi.org/10.1080/00031305.1973.10478966

[2] Justin Matejka and George Fitzmaurice. 2017. Same Stats, Different Graphs. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3025453.3025912