Reducing the Memory Footprint of Scalable Diffusion-Limited Aggregation Simulations

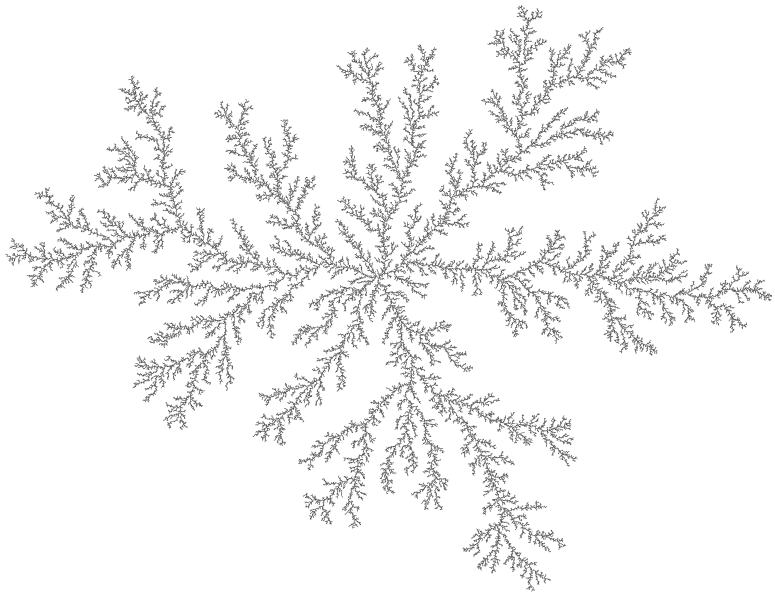
Benjamin Carter[1]

[1]**School of Physics and Astronomy, University of Nottingham, Nottingham, NG7 2RD, UK.**

ABSTRACT

Please provide an abstract of no more than 300 words. Your abstract should explain the main contributions of your article, and should not contain any material that is not included in the main text.

Keywords: DLA, Algorithm



CONTENTS

1 INTRODUCTION

Diffusion-limited aggregation is a useful method of simulation for systems where the dominant method of transport is random Brownian motion. DLA-like phenomena have been found in a variety of physical processes, such as dielectric breakdown and electrodeposition.

A basic DLA simulation consists of a single seed particle, placed at the coordinates (0,0). A particle is then released at a generation radius r_g and random angle α, proceeding to diffuse until it either sticks to the seed particle, or travels beyond a fixed kill radius r_k, whereupon the particle is reset at a new angle. This process is repeated up to a maximum number of particles, n. Without optimisations, DLA simulations are known to scale poorly. This occurs for two main reasons: First, the empty space within a DLA cluster grows faster than the Brownian trees (particles). This decrease in particle density, ρ, is proportional to the radius r such that $\rho \sim r^{D_f - D_s}$, where D_f is the fractal-dimension, and D_s is the spatial dimension, i.e. $D_s = 2$ in 2D-space. The second reason is that if the simulation is coded naively, the random walker will travel at the same small distance each iteration, resulting in greatly-increased t

To be completed

This investigation is based primarily on work by Kasper R. Kuijpers in 2014[2], where an algorithm to improve the time-scaling of diffusion-limited aggregation simulations was presented, and a more complete discussion of the algorithm can be found in the cited paper. The optimisation performs an estimation of random walker/cluster distance at each iteration of particle movement. To this end, the walker will perform large steps when at greater distances from the closest deposited particle, but to simulate Brownian motion will reduce step length on approach to the cluster. To do so, extra information (denoting the distances of each array cell from the closest particle) is stored; effectively trading an increase in speed for extra RAM usage.

The optimisation in Kuijper's work successfully reduces the time-complexity of DLA simulations to a big O notation of $O(n^{1.08})$, resulting in a (post-memory allocation) computation time of 3593 seconds for a Brownian tree of $n = 5,000,000$ particles. An unfortunate caveat, however, is that a time-complexity as low as $O(n^{1.08})$ is at its most advantageous compared to other optimisation methods at for large n particles, but the poor memory scaling of the algorithm prevents of much more than a few million particles without terabytes of available RAM. In this report I aim to present a further optimisation to Kuijpers' algorithm to retain its impressive time-scaling whilst effectively reducing the memory footprint of DLA simulations. I will also only stick to 2D simulations within this report, but will touch on methods to scale this improvement to 3 dimensions.

2 METHODS

2.1 Major Sources of Memory Usage

The large footprint of memory of the optimised DLA algorithm come from two arrays; the on-lattice distance grid Ψ, and the on-lattice cluster grid, Υ. In brief (though I recommend a complete read of Kuijpers' paper), both Υ and Ψ are 2D arrays of side length a, to be held constant during the simulation. Ψ contains the rounded-down distance of any given cell to the nearest particle within the cluster, up to a constant value D_{max}, with occupied cells denoted to have a value of 0. On the other hand, Υ contains the each particle's label, which is the order it was added to the Brownian tree, with the seed particle given the label 1. All empty sites are given the value 0. If the code needs the exact coordinates of a particle (e.g. to check potential collisions), the particle label will dictate which row to access in Ω, which is an off-lattice array

storing the exact (x, y) coordinates of the particle in question.

As the constant D_{max} will seldom need to be assigned a value greater than 255, the entire Ψ 2D array can be assigned with cells of capacity u8 (i.e. values between 0 and $2^8 - 1$, or 255), reducing the size of each cell to a single byte. It is also important to note that Ψ is integral to the operation of the optimisation–each cell within Ψ contains the necessary information for a random walker to know its distance from the cluster, and thus how far to jump next iteration. In effect, there is no wasted data within Ψ[1].

The same, however, cannot be said for Υ. It does not contain unique positional information, since if the code needs to query whether a particle can be found within array indices (i_x, i_y), it can either query if $\Psi(i_x, i_y) = 0$ or $\Upsilon(i_x, i_y) \neq 0$ and achieve the same result. Since we wish to perform DLA simulations of particles $n > 5,000,000$, it necessitates that each cell within Υ has capacity of at least u32, requiring a capacity of 4 bytes per cell. If we scale this to an array of side length $a = 40,001$ (approximately the minimum side length for $n = 5,000,000$) containing cells of size 32 bits would comprise at least 6.4GB of RAM alone. Indeed, the very nature of Brownian trees dictate that the overwhelming majority of space within the array is empty, meaning the majority of allocated RAM for this 2D matrix is wasted.

2.2 The Solution

Memory usage in this DLA optimisation can be reduced by defining an off-lattice hashmap, β; combining both the functionality of the off-lattice cluster grid, Υ, and the on-lattice array, Ω. Often referred to in other languages as a hash table or dictionary, a hashmap (or hash map) is a data structure which maps values to an associative key, typically a hash of the values to be added to the data structure. In our implementation, this hashmap β is the major source of memory optimisation. The stored values, like Ω, are the exact (x, y) coordinates of a given particle. However, in our case, the generation of the key diverges from a usual hashmap.

In a simulation, a random walker checking for potential collisions will have no knowledge of the exact positions of any particles in the cluster, only its approximate distance from the cluster and the on-lattice positions of each particle (where $\Psi(i_x, i_y) = 0$). As a result, the key in the hashmap must be both unique and based upon the on-lattice position of said particle. There are two main methods in which we can generate a key for the on-lattice position of a given particle: we can either generate a unique value based upon the array indices, or we can simply store the array indices outright.

2.2.1 Cantor Pairing Function

We can generate a unique key for the (i_x, i_y) array indices by applying Equation (1) below:

$$\pi(i_x, i_y) = \frac{1}{2}\Big[(i_x + i_y)(i_x + i_y + 1)\Big] + i_y \tag{1}$$

First described by George Cantor in 1878[1], the Cantor Equation assigns a unique natural number for any possible pair of natural numbers, in turn creating a unique single coordinate for any point in the first quadrant of the x-y plane. Since the array indices (i_x, i_y) are both positive integers, it holds that the generated key, $\pi(i_x, i_y)$ can never collide, unlike if we were to use a hashing algorithm. Whilst $\pi(i_x, i_y)$ can become large for large (i_x, i_y), within our simulations it is well below that which can stored within a u64 value. Hence, the key for each particle will be only of size 8 bytes; and the overall implementation requires far less memory than Ω and Υ within the work of Kuijpers.

[1] In the event where a random walker is outside the bounds of the array and queries its position, the code can simply return D_{max} as its current distance from the cluster, instead of returning an 'index out of bounds' error.

Should we wish to adapt this algorithm for simulations in higher dimensions, it is conceptually possible to generate a unique key for each set of $(i_x, i_y, i_z, ...)$ array indices by recursively calling Equation (1), such as is shown below,

$$\pi_d(i_1, i_2, i_{d-1}, ... i_d) = \pi\Big(\pi\Big[\pi(i_1, i_2), ... i_{d-1}\Big], i_d\Big) \tag{2}$$

and where d is the number of spatial dimensions. Yet, from inspection of Equations (1) and (2) it is clear to see that $\pi_d(i_1, i_2, i_{d-1}, ... i_d)$ will grow extremely quickly as the number of dimensions d increases. The maximum possible array size can be found by calculating the maximum possible a where $\pi_d(a_1, a_2, a_{d-1}, ... a_d)$ (i.e. all d arguments to Equation (2)) does not result in an integer overflow. which in the case of Rust would be $2^{128} - 1$, the maximum value for a u128 integer.

This method scales well for 3 dimensions, with a maximum a of $\sim 3.037 \times 10^9$; far larger than can fit in memory. However, in 4 dimensions the maximum array size falls to 55108, and to 234 for 5 dimensions. Since it would be useful to retain an algorithm which can successfully support higher dimensions, it may instead be necessary to generate a key which is simply a store of the array indices.

2.2.2 Index Storage

A simpler method of key generation would instead be to generate a vector of array indices and instead use this as the key for each particle in β, as shown below Equation (3) for two dimensions, as is shown below.

$$\pi(i_x, i_y) = [i_x, i_y] \tag{3}$$

Compared to the Cantor Pairing Function, this method would be less memory-efficient in two dimensions as we must store a vector for each particle within the Brownian tree. However, the upside of this method is the memory size of each particle's key will scale linearly as the number of dimensions is increased, as shown in Equation (4):

$$\pi_d(i_1, i_2, i_{d-1}, ... i_d) = [i_1, i_2, i_{d-1}, ... i_d] \tag{4}$$

Consequently, the choice of which key generation method would primarily be determined by how many dimensions a user would like to simulate. For the purposes of maintaining a general solution, I would elect to perform an Index Storage method as this is the more scalable of the two in d dimensions.

3 RESULTS AND DISCUSSION

The following simulations were performed on a dedicated CCX62 Hetzner server running Ubuntu 20.04. The server has a total of 48 vCPUs using the AMD EPYC 7003 chip series, and contains 192GB of RAM. As my code is capable of running simulations for different random number seeds in parallel, I have chosen to use such a powerful server so that I can average the results of each simulations across an average of 144 random number seeds, or 3 total simulations for each core. As with Kuijpers, I have not taken memory allocation into account in the computation times.

3.1 Time-scaling of Algorithm

The scaling factor for the computation times, α, where $O(n^\alpha)$, are shown below in Figure 1, along with the raw data in Table 1. These values of α for D_{max} = 50, 90, and 140 are broadly similar to those shown by Kuijpers, but not always within statistical error of uncertainty.

n \ D_{max}	50	90	140	200
100	0.0037	0.0089	0.022	0.050
200	0.0072	0.019	0.046	0.097
500	0.019	0.051	0.12	0.25
1,000	0.042	0.11	0.20	0.51
2,000	0.087	0.21	0.53	1.12
5,000	0.26	0.55	1.45	2.88
10,000	0.70	1.28	2.92	5.97
20,000	1.07	2.63	6.02	12.5
50,000	2.86	6.88	16.0	32.2
100,000	6.50	14.8	32.8	65.4
200,000	16.4	31.4	67.4	133
500,000	63.0	88.1	175	337
1,000,000	204	203	364	683
2,000,000	752	515	779	1390
5,000,000	TBA	2210	2350	3680
α	1.33 ± 0.04	1.17 ± 0.02	1.065 ± 0.007	1.028 ± 0.002

Table 1. Table of average computation time from up to n = 5,000,000 for D_{max} values of 50, 90, 140, and 200. All times shown are expressed in seconds.

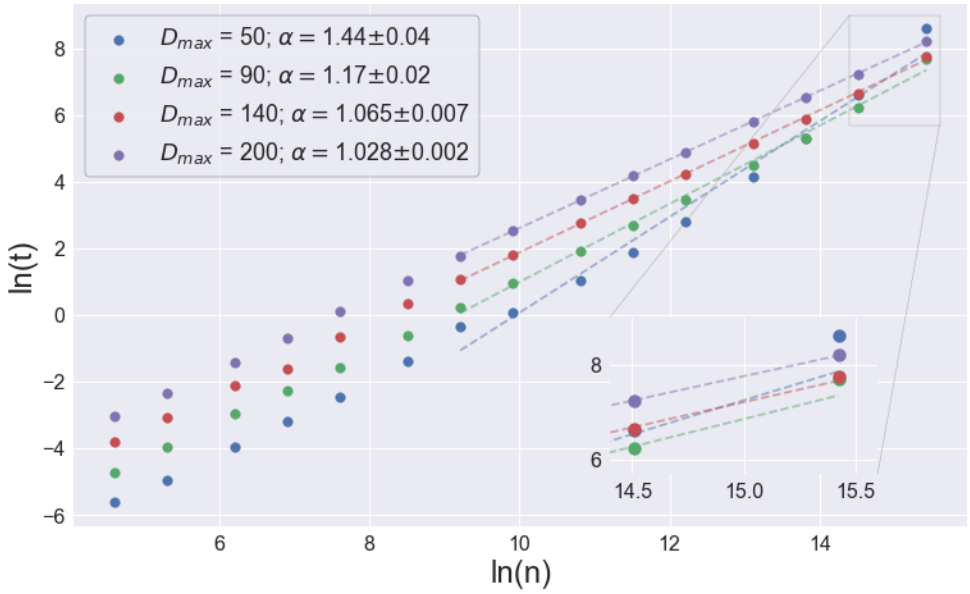


Figure 1. Scatter plot of average computation time vs. n, with a log/log scale, for D_{max} values of 50, 90, 140, and 200. Ordinary least-squares fit of the data is also shown. INSET: Zoom-in showing the extraneous entries for D_{max} = 50 and 90.

Upon reflection of the data, I am not confident that these scaling-times, α, will remain constant as n becomes arbitrarily large. For D_{max} = 50 we can clearly see that the data starts reasonably linearly, then becomes increasingly polynomial as n increases to 5,000,000. Indeed,

as shown in the inset of Figure 1, we can see $D_{max} = 90$ is beginning to curve away from the regression line. A similar non-linear trend in Kuijpers' work can be seen, but not discussed. Determining the nature of this non-linear behaviour is certainly possible, but would require further simulations at $n > 5,000,000$ and may thus require significant compute time. It may also be reasonable to suggest that with a choice of $D_{max} > 140$ that the hardware performing the simulation will have insufficient RAM before α begins to grow signifcantly.

3.2 Fractal Dimension

As shown below in Figure 2, simulations for all D_{max} values tested have resulted in fractal dimensions, d_f, between 1.714 and 1.715, in good agreement with published values[3]. The tight spread of data points for $D_{max} = 50$ would perhaps also suggest that the simulations performed for this maximum distance are not behaving anomalously. The above figure also successfully demonstrates the property of Kuijper's algorithm that the choice of D_{max} does not affect the fractal dimension, d_f, and thus the any value of D_{max} from 1 onwards should not adversely affect the nature of any simulated Brownian trees.

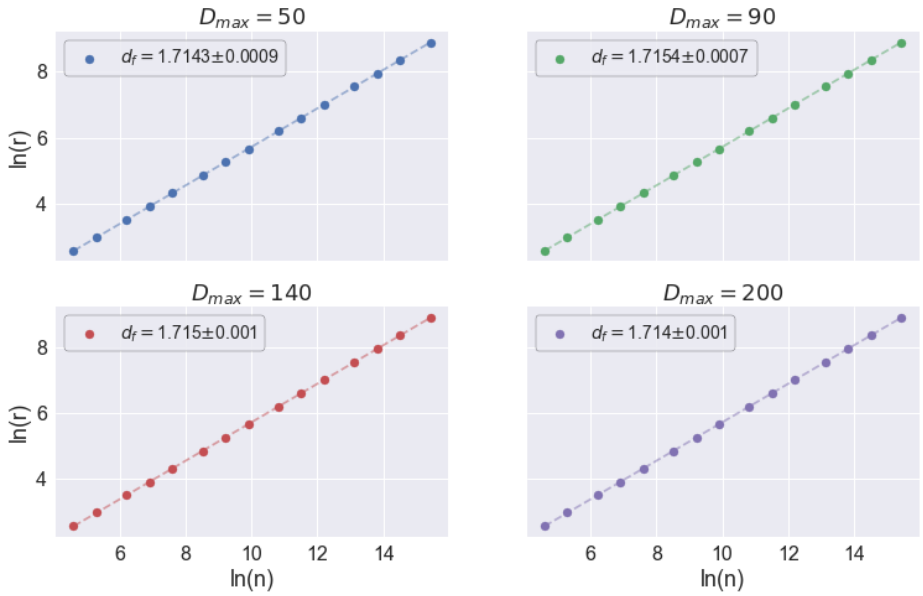


Figure 2. Scatter plots of gyration radii vs. n, with a log/log scale, for D_{max} values of 50, 90, 140, and 200. Ordinary least-square fits for each scatter plot are shown.

3.3 Reduction in Memory Usage

4 CONCLUSIONS

ACKNOWLEDGMENTS

Additional information can be given in the template, such as to not include funder information in the acknowledgments section.

REFERENCES

[1] George Cantor. Ein beitrag zur mannigfaltigkeitslehre. *Journal für die reine und angewandte Mathematik*, 84:242–258, 1877.

[2] Kasper R. Kuijpers, Lilian de Martín, and J. Ruud van Ommen. Optimizing off-lattice diffusion-limited aggregation. *Computer Physics Communications*, 185(3):841–846, 2014.

[3] Susan Tolman and Paul Meakin. Off-lattice and hypercubic-lattice models for diffusion-limited aggregation in dimensionalities 2–8. *Phys. Rev. A*, 40:428–437, Jul 1989.