Last edited on 31 January 2022

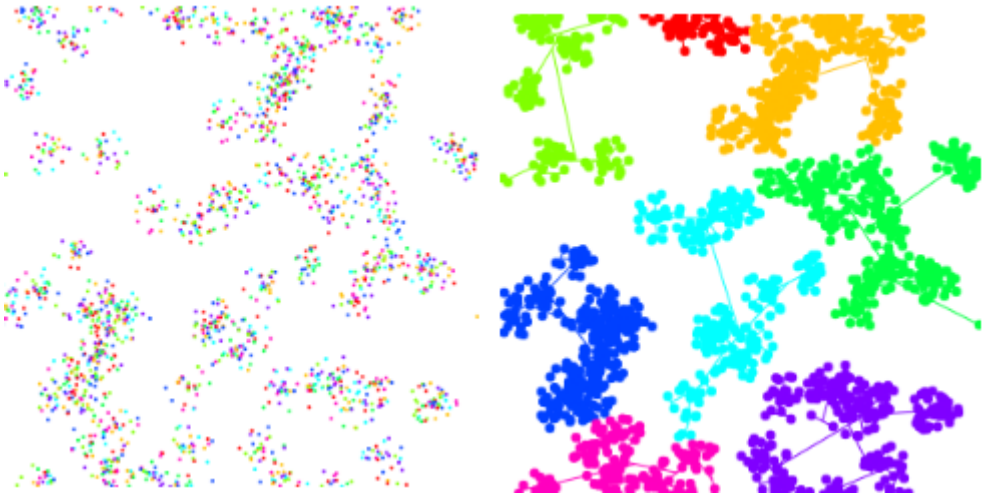
Agglomerative Hierarchical Clustering

A new algorithm using a QuadTree

Source can be found here
https://github.com/paganaye/fast-hierarchical-clustering

Wikipedia Hierarchical clustering

> In data mining and statistics, hierarchical clustering (also called hierarchical cluster analysis or HCA) is a method of cluster analysis which seeks to build a hierarchy of clusters.



This document attempts to reduce the complexity in these specific scenario:

- Points distance are measured using euclidean distances
- Clusters are linked by their centroid.

I present a new algorithm that presents the same result as the classic algorithm but runs a lot faster.
I wondered if someone would be interested enough to tell me whether my technique is already known.

Classic Algorithm

The classic algorithm is really simple (using javascript as an example):

```
function classic_hierarchical_clustering(points, wanted_clusters) {
    while (points.length > wanted_clusters) {
        let pair = find_nearest_two_points();
        merge(pair);
    }
}
```

This algorithm loops `points.length > wanted_clusters` times.
This is not too bad, but the problem is in the function `find_nearest_two_points`.
This function is extremely costly.

```
  function find_two_nearest_points() {
     let bestPair = undefined;
     for (let i1 = 0; i1 < len; i1++) {
        let point1 = points[i1];
        for (let i2 = i1 + 1; i2 < len; i2++) {
           let point2 = points[i2];
           let dx = point1.x - point2.x;
           let dy = point1.y - point2.y;
           let distanceSquared = dx * dx + dy * dy;
           if (distanceSquared < distanceSquaredMin) {
              distanceSquaredMin = distanceSquared;
              bestPair = [i1, i2];
           }
        }
     }
  }
```

This function above, saves calculating both the distance from p1 to p2 and the distance from p2 to p1. Also it only compares the square distances and saves a square root.
This is not glorious, we still have to loop through this $n^2 / 2$ times.

Overall we end up with $n^3 /2$ iterations.

Wikipedia article says:

> The standard algorithm for hierarchical agglomerative clustering (HAC) has a time complexity of $\Omega(N3)$ and requires $\Omega(N2)$ memory.

They expect the algorithm to keep a matrix of the point distances in RAM.

New Algorithm

I am attempting here to describe an algorithm that produces exactly the same results but faster.

The algorithm I imagined is quite similar upfront.

```
function proposed_hierarchical_clustering(points, wanted_clusters) {
    while (points.length > wanted_clusters) {
        if (!pairs.length) pairs = getNextPairs();
        let pair = pairs.pop();
        pairs = merge_points_and_pairs(pair, pairs);
    }
}
```

Running multiple passes.

To reduce memory usage, the algorithm runs in multiple passes.

It starts with a small given maximal distance for example 0.0001.
In the first iteration, the function **`getNextPairs`** goes through the points and returns a list of pairs whose distance is below the initial value of 0.0001.
Then the algorithm will process this list fully, merging each pair one by one.
It will then call **`getNextPairs`** again with a slightly bigger maximal distance, up to the point where we have the amount of wanted clusters.

While merging points, it will check whether new pairs below our maximal distance should be added to our pairs list.

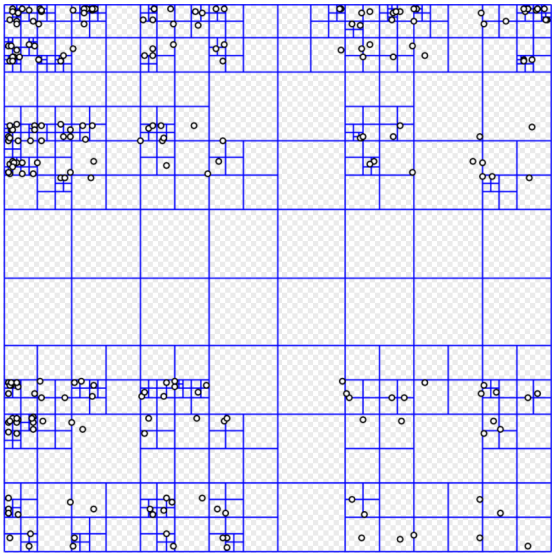
This reduces memory usage but so far, certainly not speed.

Using a quadtree

The algorithm stores all the points in a QuadTree.
A quadtree is a tree data structure in which each internal node has exactly four children.

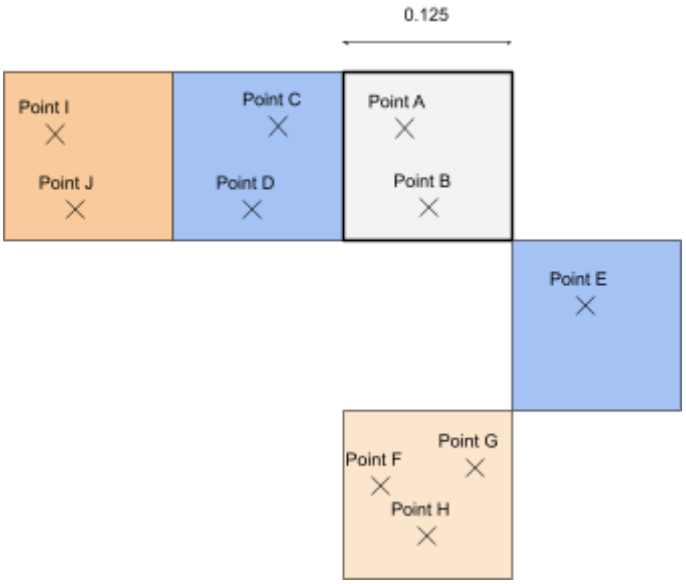
The points are all within a (0,0)-(1;1)
In its current implementation the quadtree is built with a default depth of 10 levels.

The biggest top node has a size of 1 unit whereas the smallest a size of $1/2^{10}$ = 0.0009765625.



In a QuadTree each point is stored in a container, so it is easy to exclude most pairs above a certain distance without doing any arithmetic.

For example in the sample in the picture below, if we search any pair with a distance below 0.125 starting from the white cell, we only have to visit the white cell itself and its blue direct neighbours (there are two here but, you can have, one on the top left, top, top right, left, right, bottom left, bottom and bottom right)



Using this technique getting the direct neighbours takes a fraction of the time it would take if you were building a large distance matrix.
In the example above the **`getNextPairs`** function will calculate the distances AC AD AE BC BD BE and keep all pairs where the distance is below the current maximum distance.
In the actual implementation as we are progressing slowly, empirically we can see that most of the pairs are kept.

Auto tuning

The algorithm overall speed is dependent on the maximal distance progression speed.
If it is too conservative we are going too many passes and waste a lot of time.
If it is too optimistic the list of pairs can grow very large. In this scenario the algorithm will run as badly as the classic algorithm which is probably worse.

This is why this algorithm implements a simple auto tuning of the maximal distance progression. pairs. It is really simple, if the found pairs are getting large it will slow down on the next iteration and if it is too low it will accelerate.
In the example below you can see the logs of the demo page running on a two million points dataset.
As you can see the program will try to keep the pair list between 871 and 1741 pairs.

```
Pass 13 getting pairs below 0.00000: 67/1999941 points. We're below 870.5, accelerating.
Pass 14 getting pairs below 0.00001: 163/1999874 points. We're below 870.5,
accelerating.
Pass 15 getting pairs below 0.00001: 456/1999711 points. We're below 870.5,
accelerating.
Pass 16 getting pairs below 0.00002: 1584/1999255 points.
Pass 17 getting pairs below 0.00003: 5064/1997671 points. We're over 1741. Breaking
hard.
```

Speed and complexity

I would quite like to calculate the complexity of the algorithm.
I am far more a coder than a Mathematician.
So I am publishing this first to find out if this technique is any good or known.
I am trying to calculate the complexity myself.

Formal complexity calculation is not my forte. So I am doing it my way and would love to get some feedback on it.

First, here is what I measured using a single threaded algorithm in Javascript within the demo web app.

Points	Duration (sec)
10 000	0.39
20 000	0.89
30 000	1.47
40 000	2.18
50 000	3.02
60 000	4.26
70 000	5.28

80 000	6.19
90 000	7.62
100 000	9.37
200 000	29.83
300 000	61.59
400 000	104.83
500 000	163.37

So how can we measure this?
On a given dataset the program will run a number of passes.
On each pass, we browse the entire quadtree once, extract roughly target_pairs pairs, and sort the pairs. Then for some of the pairs, calculate a new centroid, insert it into the quadtree. Then find the direct neighbours of this new centroid and add them to the list of pairs.

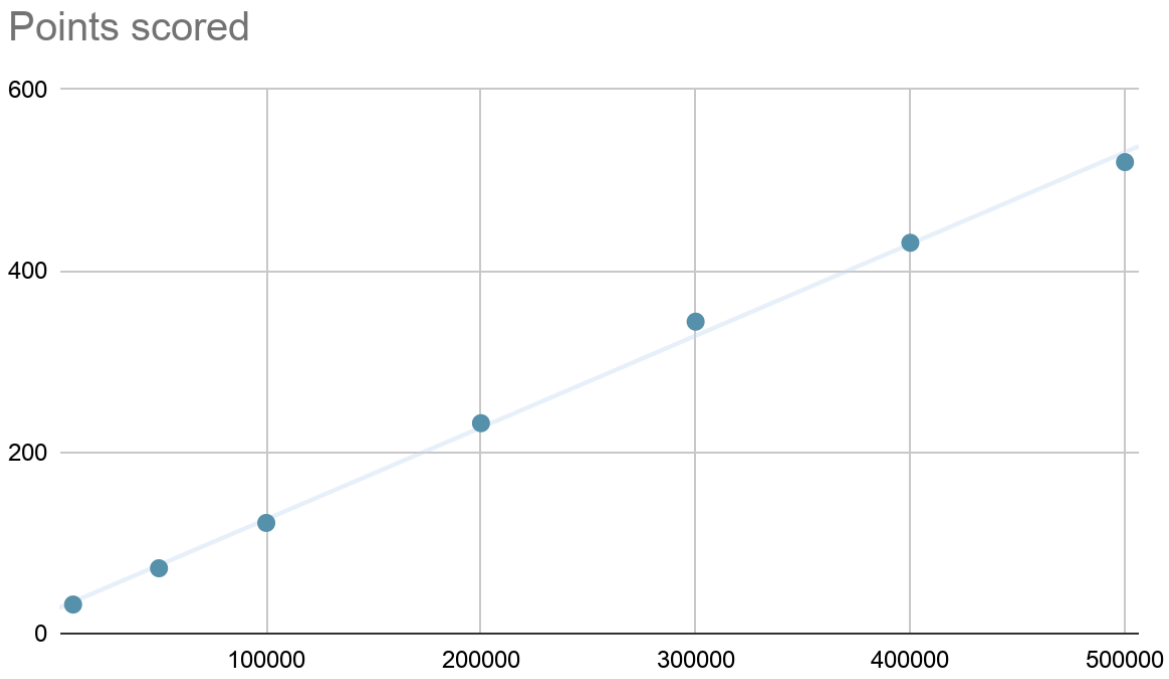
The overall complexity is

```
number_of_passes * (
        Ω(browsing_entire_quad_tree)
        + Ω(sorting(found_pairs))
        + (found_pairs * successful_pairs_ratio * count_of_direct_neighbours) * (
                Ω(inserting_successful_pairs_in_quadtree)
                + Ω(fetching_direct_neighbours_from_quadtree)
                + Ω(adding_new_pairs_to_list)
        )
)
```

We know the complexity of most of this. We can fix some of the variables and bracket others.

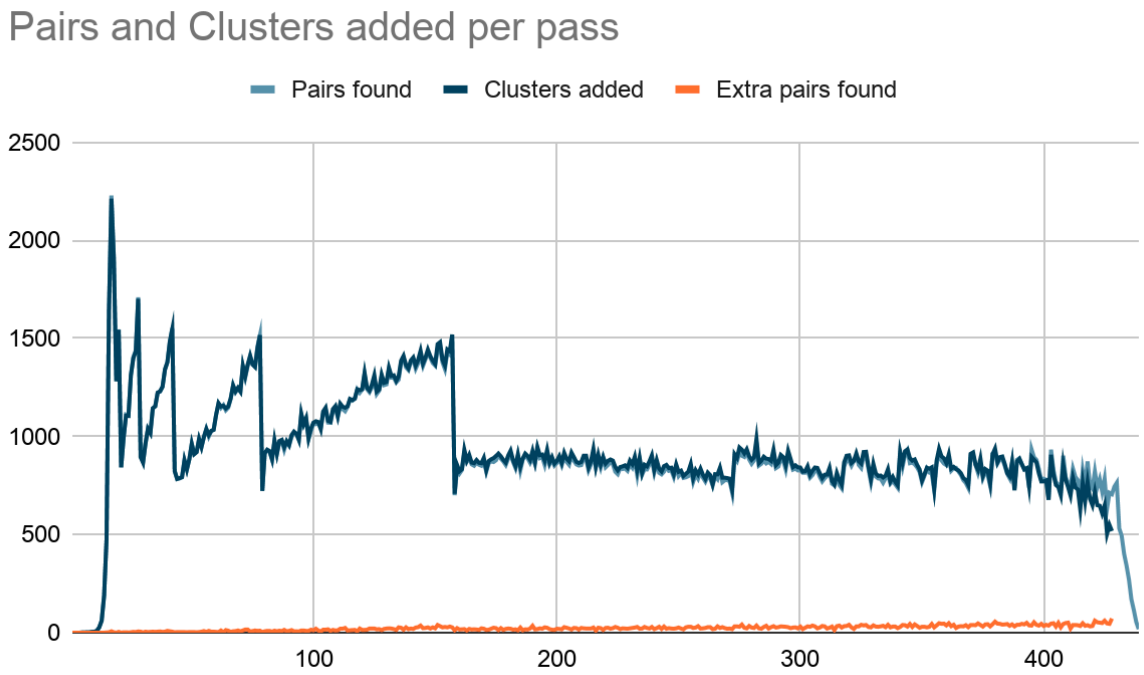
I arbitrarily fixed `target_pairs_count` to 1500 and measured how many passes the algorithm required on random data.

Points	Passes
10000	32
50000	72
100000	122
200000	232
300000	344
400000	431
500000	520



With this random data we can see `number_of_passes` appears linear and be roughly equal to `n / 1040`.

Looking at the data in more detail, we can see that the auto tuning algorithm accelerates and brakes several times in the first half time. It could be improved but I am quite happy with how well this works.



Pairs found per pass	924.3
Clusters added per pass	929.4
Extra pairs added per pass	22.5

It is interesting to see that the number of clusters added is very near the number of pairs found.
Most of the pair found end up being in the actual final Dendrogram which is pretty good.

```
number_of_passes ≈ n / 928
Ω(browsing_entire_quad_tree) ≈ remaining_points log remaining_points
Ω(sorting(found_pairs) ≈ found_pairs log found_pairs
found_pairs ≈ 930 ≈ 62% target_pairs_count
successful_pairs_ratio ≈  1
count_of_direct_neighbours ≈ 22 ≈ 1.5% target_pairs_count
Ω(inserting_successful_pairs_in_quadtree) ≈ log remaining_points
Ω(fetching_direct_neighbours_from_quadtree)≈ log remaining_points
Ω(adding_new_pairs_to_list) ≈ count_of_direct_neighbours * log * found_pairs
Ω(browsing_entire_quad_tree) = remaining_n log remaining_n
```

The overall complexity was

```
number_of_passes * (
        Ω(browsing_entire_quad_tree)
        + Ω(sorting(found_pairs))
        + (found_pairs * successful_pairs_ratio * count_of_direct_neighbours) * (
                Ω(inserting_successful_pairs_in_quadtree)
                + Ω(fetching_direct_neighbours_from_quadtree)
                + Ω(adding_new_pairs_to_list)
        )
)
```

With a number of approximation our overall could then be
The overall complexity would be

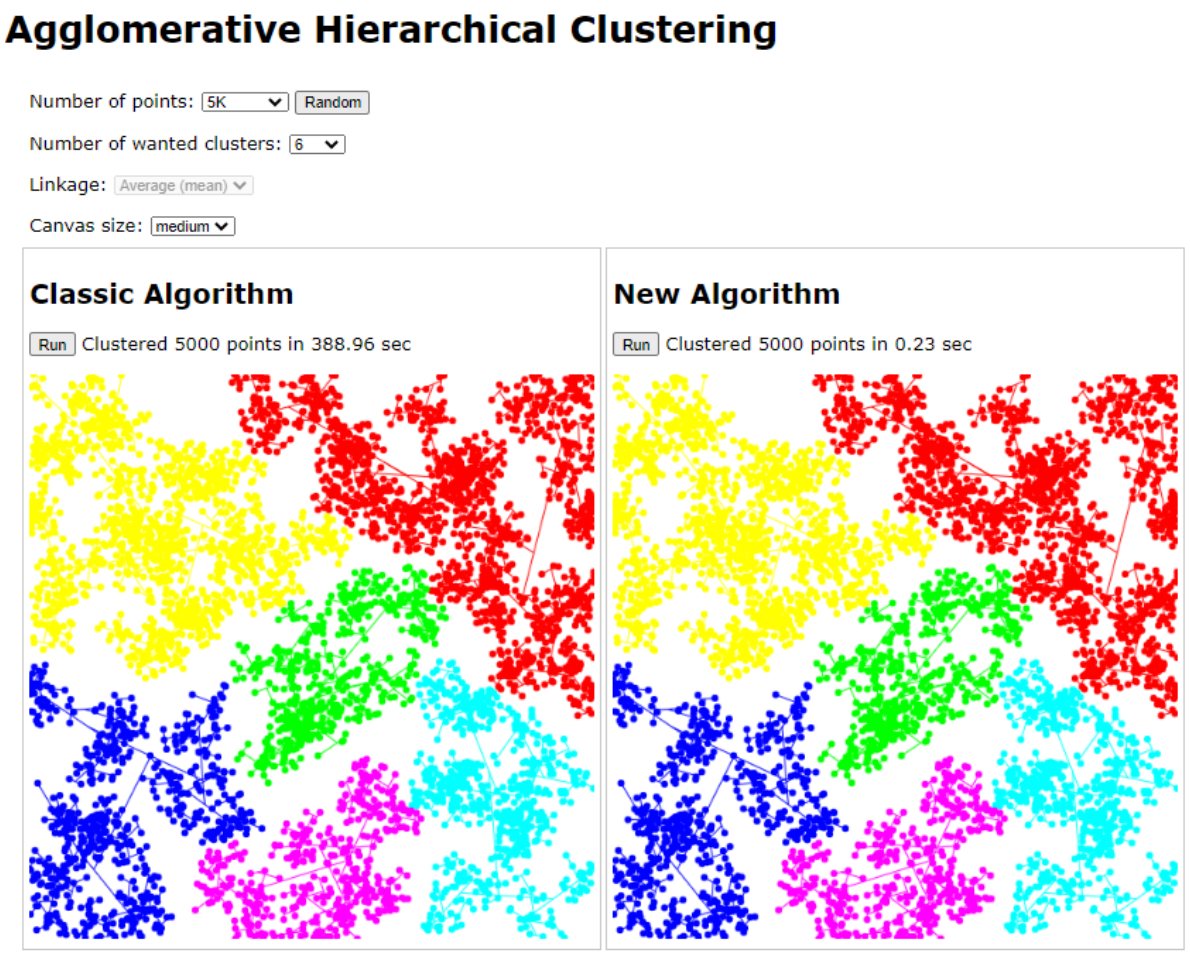
```
n * ( n log n + k) / target_pairs_count
```

Where k is a constant `≈ target_pairs_count log target_pairs_count`
So it seems to me that we end up with a complexity of $\Omega(N2)$ and requires $\Omega(N)$ memory.
Reaching my Maths limits here.

I am more at ease with the demo app which I find more convincing.

Demo App

A demo app can be found here:
http://ganaye.com/ahc



I used the classic algorithm merely to compare the result dendrogram and make sure that the new algorithm returns the same data. If I were to store a n^2 distance matrix in RAM, it would run faster but also quickly use too much memory for the browser.

You can graphically see that the algorithms return the same result.

This file's latest version is available at this address:
Agglomerative Hierarchical Clustering - A new algorithm using a QuadTree
https://docs.google.com/document/d/1d8vldwLr56vUYoXOg0B1O2WVNddBHCNKatMYUIR_atc/edit?usp=sharing