Object Oriented Spatial Algorithms
The Dorling Cartogram

s0831408@sms.ed.ac.uk

Abstract

Through object-oriented programming in python, this paper discusses different implementations of the Dorling Cartogram as a tool to visualise demographic data as spatial data. We discuss the problems faced with making the data visually pleasing and suggest how to ensure that areas do not overlap.

1 Introduction

In this report we use Daniel Dorling's algorithm and a modified version as discussed in [1], to implement a non-contiguous Dorling cartogram [2]. A cartogram is a method of visually presenting the relationship between demographic data and spatial data. They allow you to visualise the ratio between spatial relationships and data regions, while displaying the ratio of the regions physical size and data size. This results in interesting visualisations that have allowed researchers to make observations based on these relationships [4].

In non-contiguous cartograms, we attempt to minimize the overlap of the circular representation of the data in order to maintain the visual representation and to maintain readability. To do this, the region data points must be moved to allow close connections with their neighbouring regions, while maintaining as much of the original spatial data as possible. The movement of the region points can be accomplished through attraction and repulsion properties; When circles overlap, we push them apart, when there is a gap between neighbours, we pull them back together.

We wish to use Python and an object-oriented programming style to implement the algorithm described in [1, 2]. Section 2 describes the classes and functionality of each in the process of creating a cartogram and also the variables that affect the output cartogram. Section 3 shows the resulting cartograms when using a test data set of 25 regional points evenly spaced with random demographic value. We also show how effective the algorithms are when using real world data, specifically the 2005 population data for the countries of the world. We discuss what best prevents the cartograms from overlapping, and the trade offs that we incur in doing so in section 4.

2 Methodology

We should first state that the global adjacency data has been used at a scale of 0.001 in order to fit the points over the generalised geographical spatial data. The overlap of the data can be seen in figure 1.

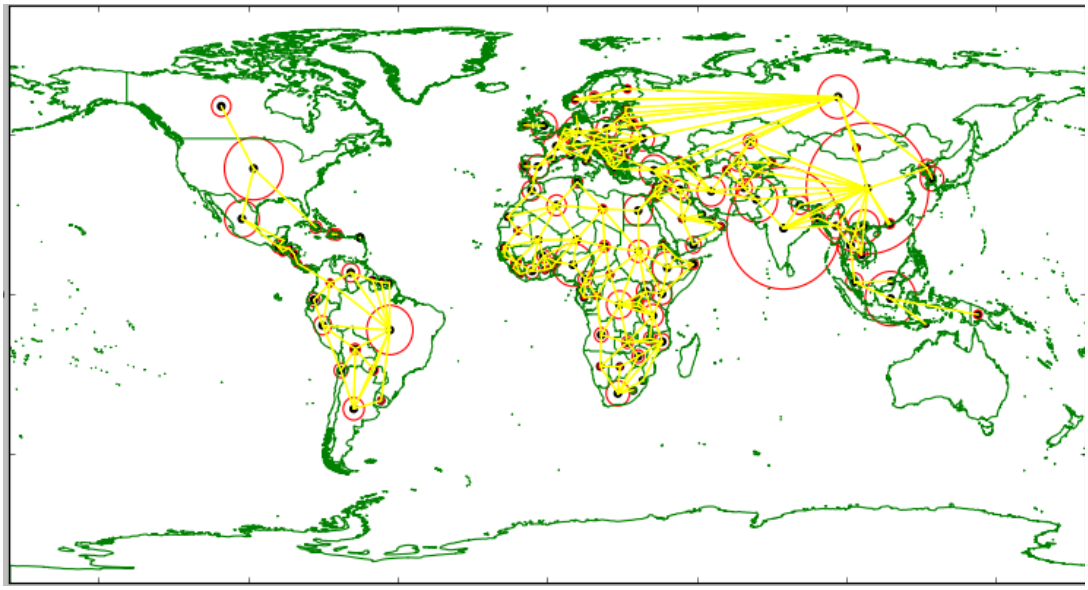

Figure 1: The overlapping geographical boundary data and population data

2.1 Movement Calculation

Using the algorithm described in [1] we assign a data value D_i to each point in the cartogram. Using the implementation for [1] we can get a radius value r_i for this point to create a circle using (1).

$$r_i = \sqrt{D_i/\pi} \tag{1}$$

Each data region needs the following information:

- Coordinates in geographical space. Used to keep the relative positions of the data regions.
- Coordinates in the cartogram. Used to keep the circles close together without overlapping.
- The neighbouring regions that share borders.
- The data value.
- The radius of the circle, calculated through the data value.
- The distances between neighbouring data regions.
- The bearing from the centroid of the data region to all neighbouring regions for both geographical and cartogram coordinates.

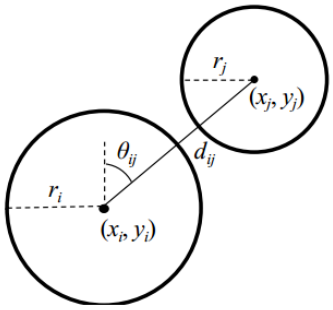

Figure 1: The measurements required between neighbouring regions

As we are implementing circular cartograms and not distance cartograms, as discussed in [2], we must avoid overlapping to ensure the cartogram is readable and each point is differentiable. For this we use the equation in (2) which, using weight value α, balances between keeping the neighbour circles close while maintaining similarity between relative positions.

$$\underset{x,y}{Min}\left[\alpha\sum_{(i,j)\in C}\left(\frac{d_{ij}}{r_i+r_j}-1\right)^2+(1-\alpha)\sum_{(i,j)\in C}\left(\theta_{ij}-\theta_{ij}^G\right)^2\right]$$
$$subject\ to\ \ d_{mn}\geq r_m+r_n\ \ \forall(m,n)\ \ m\neq n$$
$$where\ d_{ij}=\sqrt{\left(x_i-x_j\right)^2+\left(y_i-y_j\right)^2},\ \theta_{ij}=\tan^{-1}\frac{y_i-y_j}{x_i-x_j},\ \theta_{ij}^G=\tan^{-1}\frac{y_i^G-y_j^G}{x_i^G-x_j^G},\ 0\leq\alpha\leq 1 \qquad (2)$$

2.2 Implementation

We will refer to the regions as atoms that have the properties listed in section 2.1. We create these atoms as objects of the DorlinCarto class. The original implementation moved the circles in relation to their neighbours, but had no form of keeping the original spatial layout. As such we must implement the method described in section 2.1, which uses the geographical and cartogram bearings of neighbours, to keep them as close to the original map as possible.

We run everything from the *DataDriver.py* file, recieving user prompts to whether test or real data is to be made into a Cartogram and over how many iterations. It calls the *DorlingPolygonHandler.py* to either produce test data or load the real data from file and converting the data into Dorling Carto Atoms with the *DorlingCarto.py* class. With real data, we use spatial data collected from country boundaries to map the world as a comparison tool. As this data is very large and detailed, we reduce the detail using recursion through the Douglas-Peuker line generalisation algorithm.

To implement the move calculation from 2.1, the Carto_link object requires an instance variable which holds its bearings relative to its neighbours; `gBearing` or geographical bearing. The `gBearing` belongs to the link, as it is information of the specific link. We calculate the `cBearing` (the current bearing to a neighbour in the cartogram) inside Carto_atom. We treat this as a method belonging to a Carto_atom object as it this atom that is querying the information.

The *DorlingCarto.py* class, `Carto_link()`, was updated to hold the new bearing information. A getter was also needed in `Carto_link()` to access the new geographic bearing information. The bearing to a point can be calculated by using the method `bearingTo()` which is inherited from the *Points.py* super class. The geographical bearing is calculated in the constructor as the link is instantiated. The carto bearing must be recalculated every time the points move.

The new algorithm used to resolve the forces required to move the atoms is located in the *DorlingCarto.py* class, named `resolvForces2()`. The first two variable passed to the method are the same, but the third is a weighting between the two forces, α, which must be between 0 and 1.

2.3 Other Additions

It has been noted that small gaps between neighbouring circles can throw the whole cartogram apart for a few more hundred iterations until it is resolved. This takes a lot of computation, and can not be predicted as to when it will stabilize. As such we can add a tolerance level which allows a small overlap or gap between circles. This small gap should not be visible on large data sets. This tolerance level can be added as a check before the push and pull forces are executed as seen in (3).

```
tolerance = 1                                                    (3)
     if abs(distance) > (max_sep+tolerance):
          resolveForces
```

3 Evaluation and Discussion

3.1 Original Resolve Forces Algorithm

The following sequences of plots shows the 25 atom grid test environment being subject to the original Cartogram algorithm with a *push enhance* value of 20. This sequence shows 80 iterations with snapshots at 20 iteration intervals:

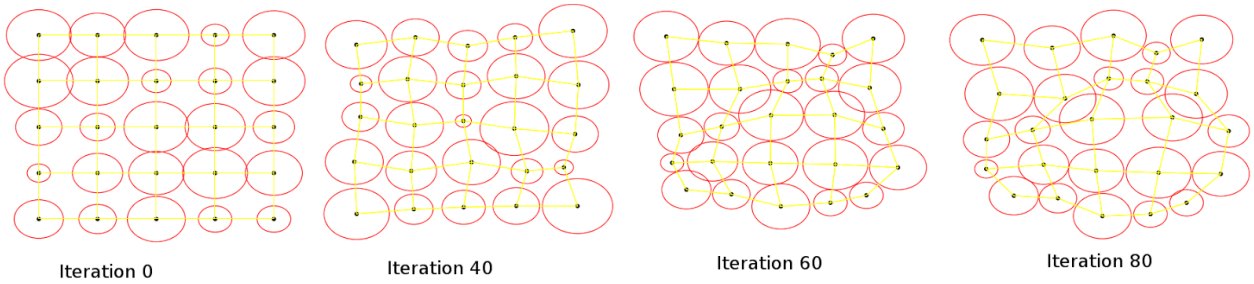

Figure 2: 80 iterations with the original algorithm, push enhance set to original 20

It is clear that at this many iterations, the force changes between each iteration are still very large, especially when there are many neighbours with very different boundary lengths. Through many trials, it was found that between 60 and 140 iterations gave the best results. The more variance between neighbouring regions, the more iterations it takes to resolve. However, once most neighbouring boundaries are touching perfectly, we notice a bounce effect where all circles push each other away again, in an attempt to resolve small gaps. This can be seen between iterations 60 and 80 above. This can be resolved by including a larger tolerance level in the max separation (see section 2.3). With a tolerance value a bit larger, we notice that the forces do not move the circles. Although this leaves a small overlap or gap, it is not visible on the data as can be seen in the image below:

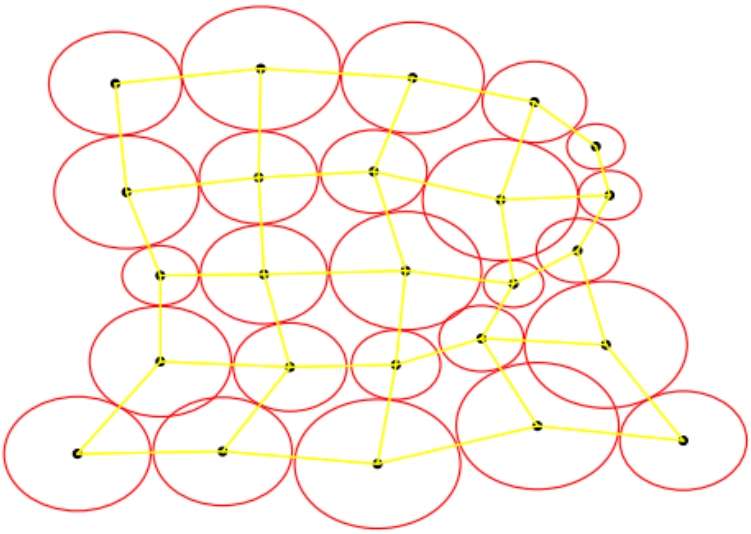

Figure 3: Results after threshold is introduced

We do however have one problem that cannot be solved so easily, and is best explained with more iterations. If we increase the number of iterations to 1,000 we see a lot less fluctuation between each iteration as the force changes are smaller, thus we can conclude that the more iterations we run, the more likely we are to have a complete cartogram. Below, each image is a plot after 200 iterations of resolve:

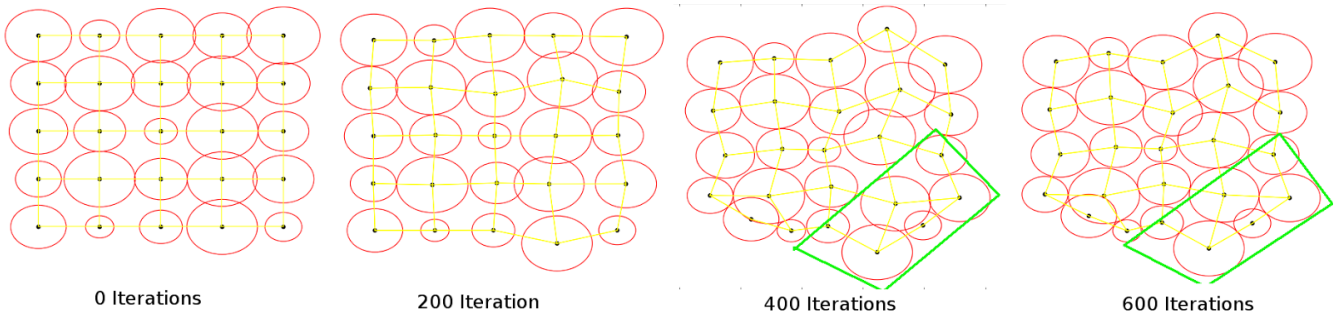

Figure 4: This figure shows the overlap of neighbour's neighbours

As can be seen, after 600 iterations, all neighbouring regions are touching boundaries with no overlaps. The problem that can clearly be seen in the highlighted sections, is that neighbours of neighbours start to overlap as the algorithm pulls all the regions closer together. After 10,000 iterations, all neighbours of neighbours are almost completely overlapping.

Alternatively, changing the *push_enhance* value from the original 20 to a smaller 5 changes the large force changes previously mentioned. A smaller push level causes the pull force to have more effect and therefore keep the regions closer together when solving minute gaps in some neighbours. With a *push_enhance* level of 5 we see the algorithm converge in 400 with no changes to boundaries after this. This however does not resolve the boundary overlap of non-topological linked points. This same effect can be seen happening with the real world data as seen below even after 100 iterations. By 1000 iterations (figure 6), the boundaries are touching, but second neighbours are still overlapping. We resolve this using bearings.

Changing the `dampingValue` in *DataDrive.py* has the effect of changing how far apart the points move with each iteration. With the value set to 0.01, the points moved very little apart in 100 iterations. With the value set to 0.5, the points rapidly move around at large distance intervals. A value of 0.1 give a good amount of movement between iterations.

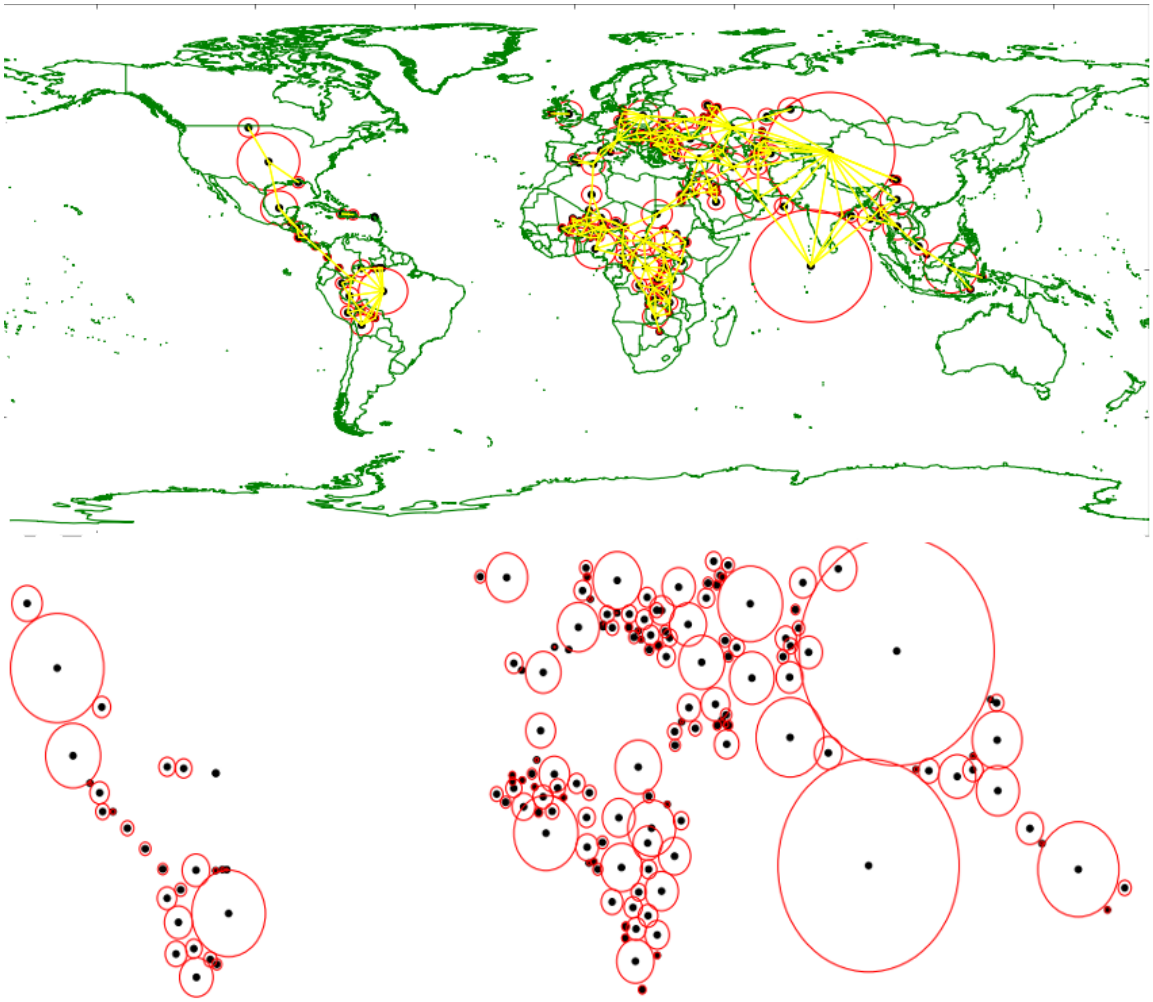

Figure 5: 100 iterations with original algorithm. Push enhance was set to 5

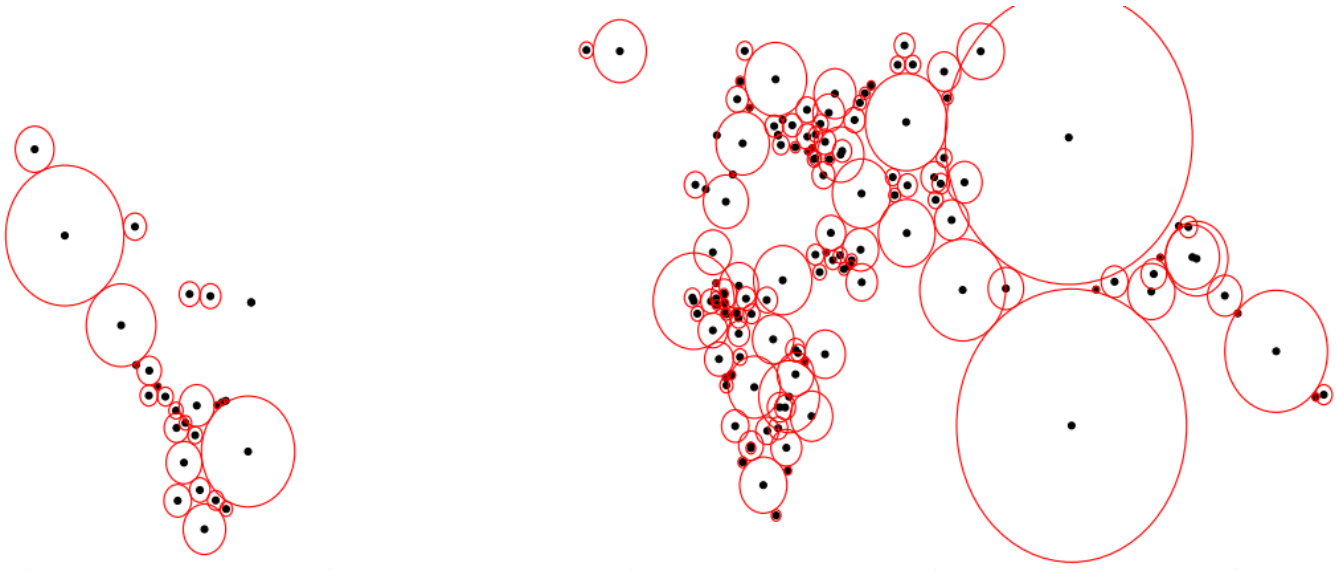

Figure 6: 1000 iterations shows the neighbours touching and non topologically linked points overlapping

3.2 Resolving Forces with Bearing

The problem of regions, not topologically linked, can be resolved using the algorithm described in section 2.1. Unfortunately, due to the algorithm described in [1], it is not clear what the "constraint conditions" between points are. Then Min x,y in (2) is unclear to its purpose. The code as described in 2.1 was implemented as described in 2.2. Once it is clear what the resulting value should be used for the algorithm should work as described in [1]. The interpretation of the algorithm in this report is as such:

1. For each neighbouring point still further away than the max distance (sum of radii), sum the following data to obtain two objective values:
    a. Find the percentage of how far away it is. Square this number
    b. Find the bearing, in radians from atom to neighbour for both current cartogram and geographical location. Square this number
2. Apply the relevant weighting to both these objectives; α to distancing, and (1-α) to similarity of locations.
3. Sum these two objective to obtain the resulting modifier for both x and y to get the new position of the atom's centre.

This clearly does not seem correct as can be seen from the resulting image in figure 6. This is the test data after 4 iterations. It is clear that the regions are moving further away with each iteration in a positive *(x,y)* direction, as to be expected.

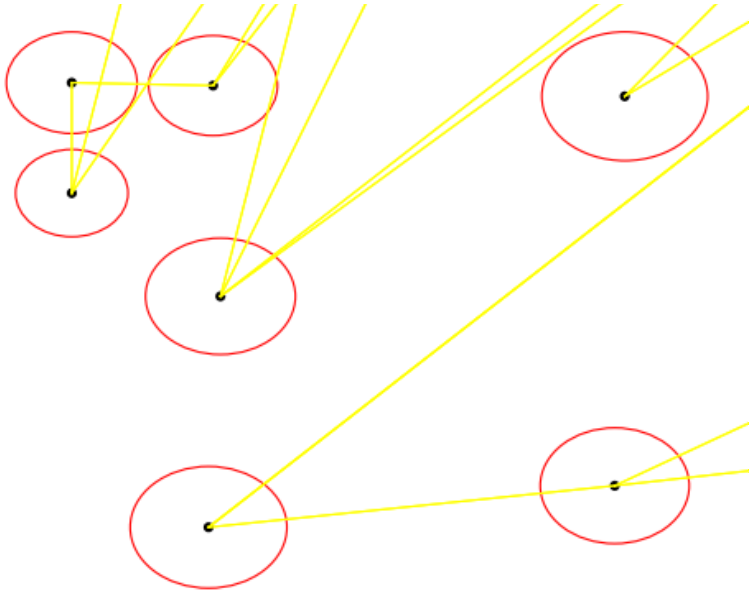

Figure 6: 2 iterations using [1] as the scaling value

4 Conclusion

The original cartogram works well for keeping topologically linked data points from overlapping, and does so relatively quickly when the correct weighting between push and pull is achieved. However, data points not topologically linked do overlap and, as seen in figure 5, can make the data difficult to read. As such, when using the original Dorling Cartogram algorithm, it is important to perform no more than 1000 iterations. The use of bearings can keep the data points closer to the original spatial layout, and prevent overlapping. The algorithm was not fully implemented due to the unclear implementation described in [1] but, with some extra understand, will work. In this paper we made effective use of object oriented design to clearly implement the Dorling Cartogram algorithm. By using classes already created and inheritance , we were able to build subclasses that inherited the methods of their superclass, preventing necessary coding and creating clear, reusable code.

5 Bibliography

1. R. Inoue, E. Shimizu, “Construction of Circular and Rectangular Cartograms by Solving Constrained Non-linear Optimization Problems”
2. D. Dorling, 1996, Area cartograms: Their use and creation. Concepts and Techniques in Modern Geography (CATMOG), 59.
3. E. Shimizu and R. Inoue, 2009, A new algorithm for distance cartogram construction. International Journal of Geographical Information Science, 23(11): 1453–1470
4. Keim, Daniel A., et al. "Efficient cartogram generation: A comparison."Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on. IEEE, 2002.