E8 Cordless Phones Report

By Megan Hayes (3996884), Kiri Lenagh-Glue (2593116), Nikolah Pearce (6377167), and Megan Seto (1227523)

Table of Contents

Foreword

To begin with, we initially brainstormed ideas on how to approach the given problem to solve this etude and discussed the benefits and inefficiencies in each basic strategy. We initially prepared by setting up .java files to read the basic input and set up a Telephone class with data fields X and Y points respectively. For lack of any better ideas on where to start, we also coded a basic Graphical User Interface to scatter the points on screen for us to visualise.

The Town Hall on April 10th brought many ideas to light that we had not yet considered. Two of the biggest hints taken from the Town Hall were:
- Use trees (always try and implement a data structure you know works efficiently and well).
- Sort each X and Y values separately (as if they were in one dimension only).

Thus, we decided to take a different approach to the problem with those two key points in mind. This ultimately lead us to k-dimensional trees (k-d tree).[1]

Another idea we had been playing with was how to *maximise* the range of 11 phones. An idea we kept throwing around was to minimise a circle enclosing the 12 most close-together phones, then reduce that circle so as to exclude one more phone. Would this not then be the maximum range of only 11? This allowed us to translate the problem from finding the maximum of 11 phones, to finding the minimum of 12, and performing some final adjustments.

As the demonstrators hinted that it is ok to utilise a library of code in order to achieve a task, we started researching an implementation of k-d trees in Java (our most confident language). This was a fruitless endeavour - as every implementation we found was either impossibly complex and poorly commented, or didn't compile and run as per the readme.txt file.

In order to avoid implementing our own k-d tree class, we then decided to switch from coding in Java, to Python. The majority of implementations we found were in other programming languages, and we easily found a Python library of a basic implementation of a k-dimensional tree[2]. This library also had methods for searching and finding the nearest neighbour of a point, and the nearest k-neighbours of a point. We were then able to import the library and set up our data structure easily.

Procedure

K-Dimensional Trees

For the given problem, we thought that k-d trees were an appropriate data structure to use, as they hold a variety of useful applications that we could exploit in order to satisfy the tasks necessary to complete the etude. Especially in any sort of efficient manner. Such applications included:
- K-d trees are space-partitioning data structures that are used to organise points in *k*-dimensions. This is a useful structure for us, as this enabled us to split on 2 dimensions - the input x and y coordinates (data inputs for the east north and points respectively)

---

[1] First understanding of this data structure came from https://en.wikipedia.org/wiki/K-d_tree
[2] The library we imported was found at https://pypi.org/project/kdtree/

- The nearest neighbour search - a search to reveal the point closest to a specified base point within a set. The search would be implemented using our 2 dimensions, x and y. We thought to extend this search to find the *k-nearest neighbours*.
- Implementation of a k-d tree is similar to a binary tree, and thus has a good search time (Etude requirements specified that the speed of the program will be an issue).

To set up the k-d tree using the chosen library, we simply had to create an instance of the class and pass our points to the constructor, either as a list or a tuple. The readme.md file for the library was particularly helpful in getting started. There were also a few helper methods included that allowed us to confirm the data structure was set up correctly and balanced etc.

Nearest K-Neighbours Search

We then invoked the search method to find the k-nearest neighbours for each and every one of our telephone sites/points in the total set. This was conveniently already included in the k-d tree library. We used a variable `max_phones` to store an integer one more than the total number of phones allowed within the maximum range i.e. 12 in this case (11 + 1 = 12). We passed this variable to the k-nearest neighbours method, and used the point of a telephone site in our set as the base point of where to search for the k-nearest neighbours from. This created a total group of 12 nearest neighbours, as the algorithm for k-nearest neighbour search would always compute a distance of 0 (the closest of all) to a point that is itself - automatically including that point, in the group of k points it returns.

We chose to search for the k-nearest neighbours for each and every point within our set. One of these sets of 12 found would have to be the minimum of all, and in order to find which, we first translated the points into a smallest enclosing circle using Welzl's algorithm.

Welzl's Algorithm (Smallest Enclosing Circle)

In order to find the optimum minum, we decided to loop through each point and invoke the `find_circle()` method from the `smallestcircle` class on the set of 12 nearest neighbours found. This then created a smallest enclosing circle of the k points input to it. One for each set of 12 we passed. We had some initial concerns about inefficiency using this method, but first we wanted to see if it even worked. The `find_circle()` method returned the center and radius of the enclosing circle, to which we then appended into a list variable to keep track of all the smallest enclosing circle data found.

Finding the Minimum Smallest Enclosing Circle

We simply looped through our set of all data stored for each smallest enclosing circle, while keeping a variable outside the loop holding the `minimum_radius`. Whenever a smaller radius of a circle was found, the minimum was swapped out and stored in the varable. As the list of all smallest enclosing circles was a 2D array in the form `{{x, y, r}, {x, y, r}}`, we were also able to pull out the x and y coordinates alongside the radius - which helped greatly when it came to testing.

Turning the Minimum Circle into the Maximum Range

At this point we had the true minimum circle enclosing 12 points, however, we needed to transform this to a maximum of 11 points. Our first and initial thoughts were to decrease the circle

found by some arbitrarily small amount (which in itself was difficult to justify). Instead, let's consider what constitutes a telephone being within range. Hear we are assuming that any telephone enclosed within our given circle is within the range. What about telephones that fall exactly on the edge of this circle?

The circle that Welzl's algorithm finds and returns to us is by definition the minimum enclosing of 12 *and* the maximum enclosing of no more than 11. One of the properties of the smallest enclosing circle is that it will have at least 2 points exactly on the circumference. The circumference needs to enclose at least two points in order for it to be the minimum. The reason that we can say the minimum of 12 is also the maximum of 11, is in the definition of a point being 'in range'. If we assume that the edge telephones are in fact not part of the range, then we have our answer. This being that we are defining being on the edge of the circle, as not the same as being within the circle. Therefore, any telephone point exactly on the edge of the circle is not in range.

We then decided to simply format the range to be to 2 decimal places (rounded down), similar to the input variables, and output this in the unit of metres (the same as indicated in the etude outline). We rounded down in order to maintain accuracy of the minimum 12.

~~We did this by decreasing the radius found by an arbitrary small number. This became the most debated part of the group etude. How do we justify the *maximum* range? We knew that the value of the range was too big to be the maximum, because there was a 12th point lying on the circumference of the circle. Hence, we needed to decrease the circumference exactly enough to exclude this 12th point. (Note: any point on the circumference can be considered the 12th point, due to the properties of the smallest enclosing circle - it does not matter *which* point we exclude so long as we exclude one that is on the circumference. Part of this logic comes from the fact that when two points opposite each other are on a circumference with radius x, the same two points can never be on a circumference with radius < x - it is simply impossible[3]). The final thing to consider here is that because we are working with a radius and not a diameter, whichever arbitrary number we choose to decrease by will be decremented from the range twice (once for the left side radius, and once for the right side radius). Thus, taking this into consideration, whichever value we choose to satisfy this number, we then need to halve it.~~

~~We chose to decrease by 0.02 metres, or 20 cm. This meant we would subtract 10cm or 0.01 metres from the range found (0.02 * 1/2 = 0.01 metres). The justification behind this value came from a number of factors~~
- ~~Our estimate of the size of a telephone was around 10 cm²~~
- ~~A telephone would not generally be larger than 20 cm² in size then, give or take~~
- ~~Decreasing the range to exclude the physical telephone size, satisfies that that phone is no longer in the range.~~

Testing

Initially, instead of finding all smallest enclosing circles of 12 points, we tried to be slightly more efficient and find the minimum distance of the 12 nearest neighbours together and then

---

[3] The smallest enclosing circle will always have two or more points on the circumference
https://en.wikipedia.org/wiki/Smallest-circle_problem

translating this to the minimum smallest enclosing circle. Unfortunately, we found this initial approach actually only found the smallest enclosing circle of the 11 closest neighbours to a specific 1 point (a group of 12 total) - not the true minimum smallest enclosing circle of 12 out of all points in the set. We discovered this bug by superimposing our output points x and y of the given circle's centre, along with its radius, over top of our plotted input points.[4]

Figure 1:
A smallest enclosing circle of the input's 12 closest together points. This includes every point except the set { A, B, E }. Radius given was 79.62 metres (2 dp). This is based off point C and it's nearest 11 neighbours which are the minimum set of nearest neighbours.
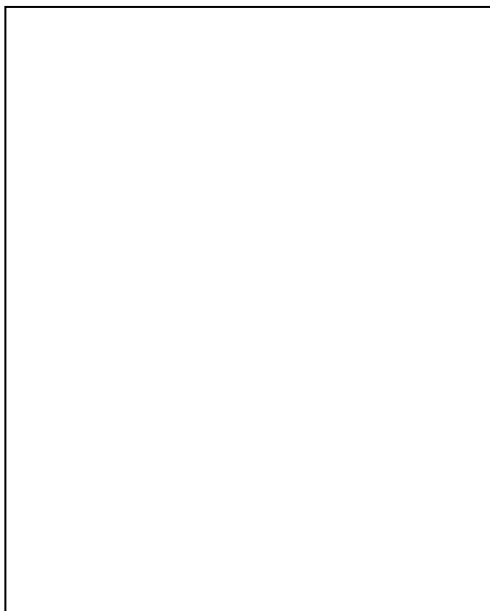

Figure 2:
We have relocated the same circle as in Figure 1 above. As you can see there are now 13 points within (every point except the set { E, D }). This is clearly then not the true minimum smallest enclosing circle of 12 points, as if it were, there would be no instances where more than 12 points could be within the circle range at any time.

What we needed to do was find the 12 closest points in the entire set of points N - not just the 11 closest together to one point. This is also known as the "smallest k-enclosing circle" problem.[5] In order to achieve this, we changed where we were optimising the 'minimum'. Instead of passing the minimum set of 12 closest together points through to the smallest enclosing circle method, we instead

---

[4] Viewing the output graphed at https://www.math10.com/en/geometry/geogebra/geogebra.html
[5] Outlined in the article https://www.sciencedirect.com/science/article/pii/0925772194900035

passed every single set of 12 nearest neighbours (one for each and every point in our graph). We then kept track of the output centre points and radius of each circle found in a list. At the end, after each point had found its 11 nearest neighbours and had a smallest enclosing circle found, we looped through the set of all smallest enclosing circles to find the true minimum.

As hoped for, the program then produced an answer of a slightly smaller radius than the original method, while still using the same input data.
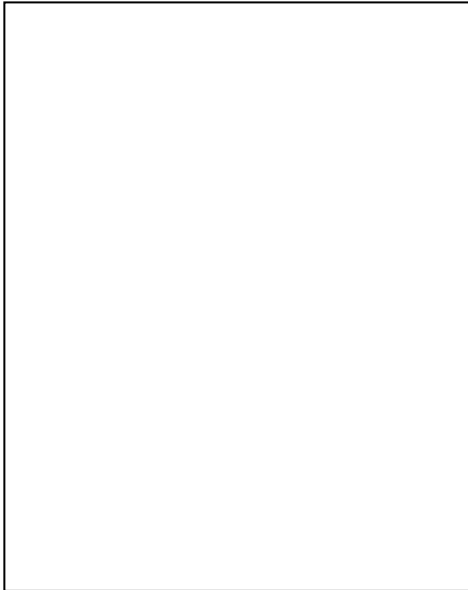
Figure 3:
The true minimum smallest enclosing circle of only 12 points. This includes every point except the set { E, D, G }. Radius given was 71.31 metres (2 dp). We know this is the true minimum enclosing circle as no matter where you relocate the circle, you can never enclose more than 12 points.

Figure 4:
After reducing the true minimum smallest enclosing circle by our arbitrary number of 20cm, we can see that we have successfully reduced the range by just enough, such that the circle now encloses a maximum of 11 points. Refer to Figure 5, Figure 6 and Figure 7 to examine the points on the outer edge of the circle.
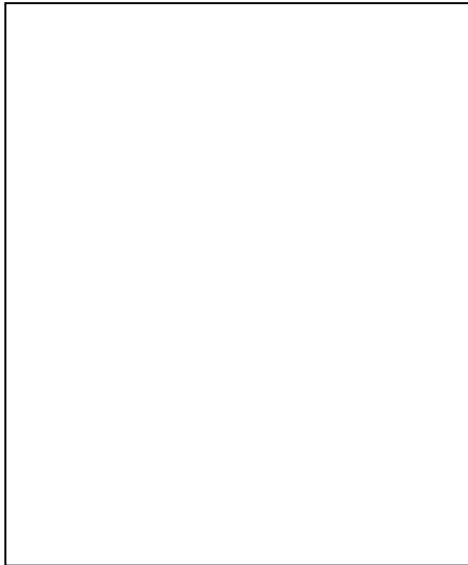
Figure 5:
When zooming in on Point K, it can be seen that it is in fact now not included in the range. This will reduce our total number of points within the range to 11.
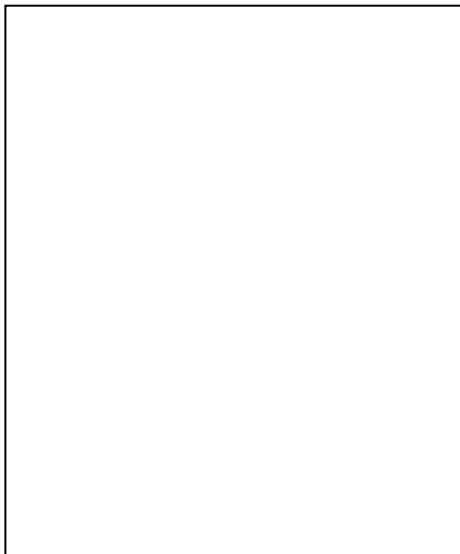
Figure 6:
When zooming in on Point G, it can be seen that it is still within the range. This has no effect on the number of points included, and it remains at 11 still.
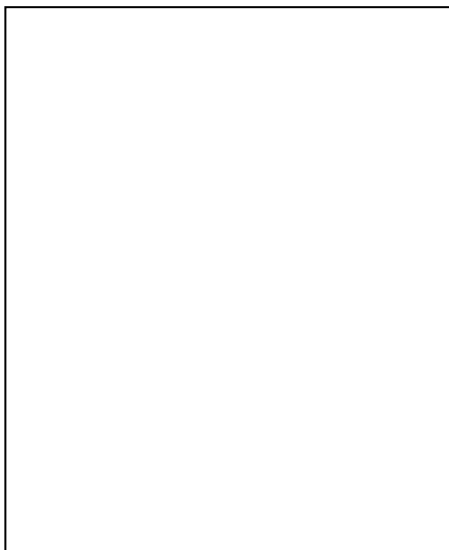
Figure 7:
When zooming in on Point B, it is also seen within the range. This again has no effect on the number of points included, and it still remains at 11.

From the above images, we concluded our program was executing as desired. No matter where we move this circle to now, it cannot contain more than 11 phones. This is because Points B and G are still on the circumference, so if we were to move the range to again include Point K, one of Point B and G would be pushed out of the range (this is because the radius is fixed and does not change).

References

1. https://en.wikipedia.org/wiki/K-d_tree
2. https://pypi.org/project/kdtree/
3. https://en.wikipedia.org/wiki/Smallest-circle_problem
4. https://www.math10.com/en/geometry/geogebra/geogebra.html
5. https://www.sciencedirect.com/science/article/pii/0925772194900035