Maximizing the Minimum Distance Between Two Convex Hulls

Duncan Greene

December 13, 2023

Abstract

The primary focus of this paper is to develop a Python algorithm that efficiently partitions a two-dimensional dataset into two non-empty subsets, maximizing the distance between their convex hulls. Drawing inspiration from "Computational Geometry" by de Berg, Cheong, van Kreveld, and Overmars, I use clustering techniques and advanced algorithms to do so. My strategy involves a unique set-partitioning technique coupled with existing convex hull packages to simplify implementation, with the goal of creating adaptable Python code for datasets of varying sizes. The motivation extends to practical applications such as robotics/path planning, support vector machines, biomedical imaging, and resource allocation clustering. To enhance efficiency, I'll tap into Kaggle as a crucial resource, utilizing Python templates for a professional workflow. This paper discusses and contributes to Computational Geometry Theory, addressing broader needs for robust partitioning solutions across various fields.

1 Introduction

This project investigates a computational geometry problem: devising an algorithm to optimally partition a set S of n points in a 2-dimensional plane into two subsets such that the distance between their respective convex hulls is maximized. The study focuses on two-dimensional spatial data, leveraging the construction and optimization of convex hulls to achieve this goal [1]. This paper explores the proposed algorithm, which utilizes various Python packages and online resources to achieve the minimum distance between two distinct convex hulls, iterating over all possible convex partitions of a set S to identify the maximum distance. The goal is to generalize the algorithm to handle any set S with any number of points n.

The paper provides a detailed walkthrough of the algorithm, discussing its design, implementation, and the challenges encountered. Additionally, it examines the algorithm's complexities and limitations, highlighting areas for potential improvement. Practical real-world applications of the algorithm are also demonstrated to emphasize its relevance.

To gain deeper insights, the algorithm is tested on diverse datasets of varying sizes, all confined to a 2-dimensional plane. This analysis offers a comprehensive understanding of the algorithm's components and their interplay. The model is designed with flexibility, allowing for assessments of its adaptability, scalability, and computational complexity. In summary, my paper seeks to elaborate on the intersection between my project, spatial data analysis, utilization of convex hulls, computational geometry, and the practical world. The following sections of this paper present the details of my algorithm and the methods used in its development. I then apply this algorithm to data sets varying in size to test and walk through the time complexity of the findings. Finally, I aim to explore the practical application of computational geometry and the distance between two convex hulls in fields like robotics, biomedical imaging, and resource allocation.

2 Algorithm Context and Walkthrough

The convex hull of a finite set S is the smallest possible convex polygon that encloses S in the plane. Convex hulls are fundamental objects in the field of computational geometry and algorithms that efficiently compute them have numerous applications in the real world. These include the domains of pattern recognition, image processing, and spatial analysis. The algorithm uses several Python packages including *matplotlib, numpy, scipy.spatial.convexhull*, and *time*.

2.1 Libraries Used

Matplotlib is a plotting library for creating static, animated, and interactive visualization in Python. The algorithm uses the '*plt.scatter*' and '*plt.plot*' functions from the *matplotlib* library to create scatter plots and visualize the computed convex hulls in numerous different ways [9]. *NumPy* is another powerful library that is used for numerical operations in Python. Though the project remains in 2D, *NumPy* provides support for large multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays [6]. In the algorithm, *NumPy* is used to generate random coordinates, create arrays, and perform various array operations. For example, the function '*np.random.rand*' generates random numbers, and '*np.array*' is used to convert lists to *NumPy* arrays [6]. Next the code uses the *SciPy* library to both compute the convex hulls using a short cut algorithm in the package named '*ConvexHull*' under the *'scipy.spatial.convexhull'* class [7]. I later use the library again to compute euclidean distance between vertices and points on a line between two vertices using the function '*euclidean*' [10]. Lastly, I im-

plemented the '*time*' module in Python to compute the computational time it takes to run numerous different examples and test the limits of the algorithm. The code for the imported packages can be seen below.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.spatial import ConvexHull
from scipy.spatial.distance import euclidean
import time
```

2.2 Partitioning Convex Hulls

After implementing a number of different libraries to help compute the distance between convex hulls later on, I began by generating a random set of points in the XY plane. For my project I choose n (number of points) to be 15 to start. In order to create repetition and test different theories on a random set of points, I implemented a seed function so that I was able to repeat the code with the same random coordinate points. Next it sorts these points based on their x-coordinates, breaking any possible ties using the y coordinates. The sorted points are subsequently used to create lines between all pairs of points. For each line, the algorithm filters the points into two groups - those lying on the left side and those that lie on the right side of the line. Points lying on the line are included in the convex hull. The question calls for a set of S points to be broken up into two subsets. In order to tackle this problem one must come up with a way to split the set S. By choosing two random points, I create a natural split line and can implement this on every combination of two points. The following pseudocode walks through this piece of the algorithm, from creating a random number of points on the plot to filtering the points based on if they lie on the left or right side of the created line. It is then followed by Figure 1 in which it shows the scatter plot of the plotted lines with the line dividing the set S into two subsets:

1. Generate random coordinates for a set number of points

   ```
   num_points = 15
   x_coordinates = generate_random_coordinates(num_points)
   y_coordinates = generate_random_coordinates(num_points)
   ```

2. Create a scatter plot of the random points

   ```
   plot_scatter(x_coordinates, y_coordinates)
   ```

3. Create a list of points using the generated coordinates

   ```
   random_points = create_points_list(x_coordinates, y_coordinates)
   ```

4. Sort the points by x-coordinate

```
sorted_points = sort_points_by_x(random_points)
```

5. Initialize variables to store maximum distance and information

```
max_distance = 0
max_info = None
max_plot = None
```

6. Loop through all combinations of two points

```
for i = 1 to num_points:
    for j = i + 1 to num_points:
```

7. Choose a line using sorted points

```
point1, point2 = choose_line(sorted_points[i], sorted_points[j])
```

8. Filter points on the left side and right side of the line

```
left_points, _ = filter_points_by_side(sorted_points, point1, point2)
right_points, _ = filter_points_by_side(sorted_points, point1, point2)
```

9. Add points lying on the line to both sides

```
on_line_points = find_points_on_line(sorted_points, point1, point2)
left_points += on_line_points
right_points += on_line_points
```

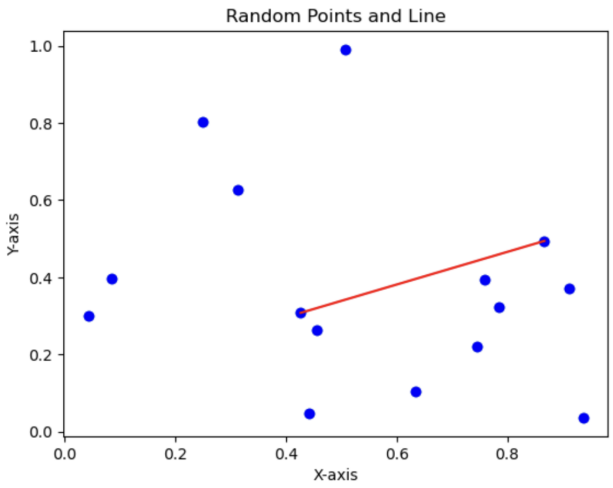


Figure 1: *Scatter Plot*

2.3 Computing the Convex Hull

Next the algorithm computes the two convex hulls for the points on the left and right side of the line between the two chosen points. It is important that the convex hulls are only calculated if the number of points on the left side of the line is three or more. The 'ConvexHull' algorithm utilizes the 'QuickHull' method, which employs a divide-and-conquer strategy to efficiently compute the convex hulls [7]. The 'QuickHull' method essentially takes the new set of points and finds the two points with the maximum and minimum x-coordinate. These will be on the convex hull. It then partitions the remaining points into two subsets: those to the left and of the line formed by the two extremes and those to the right [8]. This is then recursively applied to the two subsets where it is finally merged to complete the full convex hull. The pseudo code that follows is a piece of the algorithm that lies in the nested loop portion that creates the convex hulls using the quickhull algorithm for every combination of two convex hulls. One of these combinations of convex hulls can be seen in Figure 2.

10. Compute convex hulls for left and right points

```
hull_left = compute_convex_hull(left_points)
hull_right = compute_convex_hull(right_points)
```

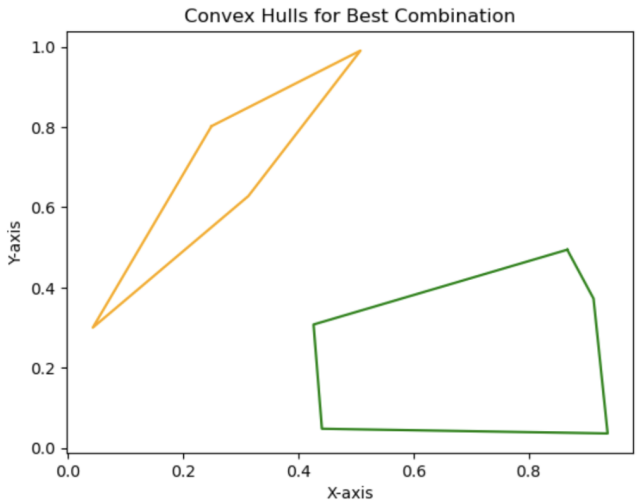


Figure 2: *Convex Hull Example*

2.4 Measuring Distance Between Hulls

Finally, the distance between the 2 convex hulls needs to be measured. In this case the algorithm implements a custom function named *'min_distance-between_hulls'*. This function takes the two '*ConvexHull*' objects and the corresponding set of points on each

side of the randomly chosen line. The algorithm then computes the distance between the two convex hulls considering different possibilities such as the minimum distance between convex hulls being the distance between two vertices on corresponding convex hulls or the distance between a vertex on one convex hull and the edge on the other convex hull. The distance measurement utilizes the Euclidean distance formula provided by the *'scipy.spatial.distance'* library [7]. The pseudo code below contains the function that computes the minimum distance between the convex hulls and stores it in a list. It then plots the convex hull with the maximum minimum distance between them which can be seen for the example in Figure 3.

11. Check if convex hulls are computed

```
if hull_left is not None and hull_right is not None:
```

12. Compute and store the minimum distance between convex hulls

```
min_distance, min_type = min_distance_between_hulls(hull_left, hull_right,
left_points, right_points)
```

13. Check if the current minimum distance is greater than the stored maximum distance

```
if min_distance > max_distance:
   max_distance = min_distance
   max_info = "Points " + i + " and " + j + ": " + min_distance
   + " (" + min_type + ")"
   max_plot = (left_points, hull_left, right_points, hull_right)
```

14. Check if the current minimum distance is greater than the stored maximum distance

```
if max_plot is not None:
    left_points, hull_left, right_points, hull_right = max_plot
    plot_convex_hulls(left_points, hull_left, right_points, hull_right)
else:
    print("Skipping convex hull computation due to insufficient points.")
```

The algorithm I have created is made to contain a nested loop so that it can explore all combinations of pairs of points, computed convex hulls and measuring distances between them. It then records the maximum distance and the elapsed CPU time for the entire process so that I can find a point where the number of points proves the algorithm inefficient.

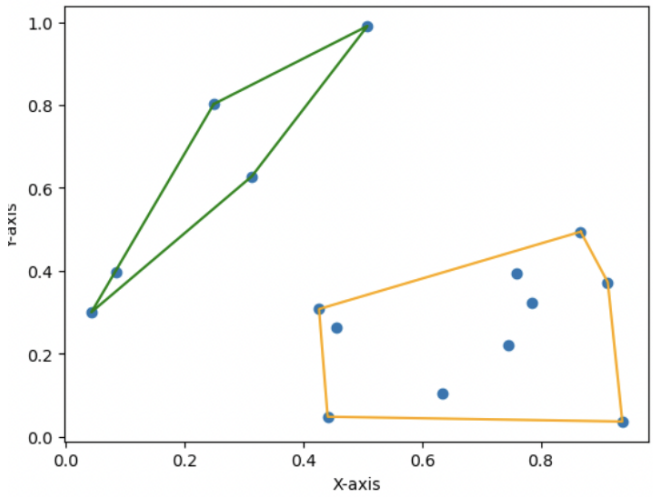
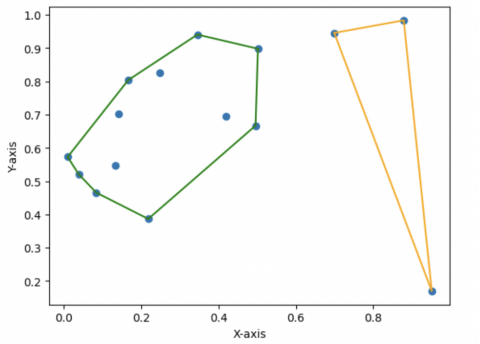


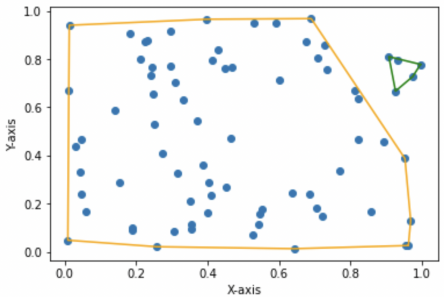
Figure 3: *Convex Hull Example with Points Plotted*

3 Results

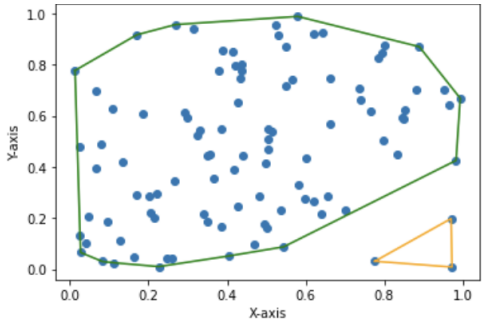
This algorithm was tested on numerous different data sets containing points between 15 and 115 points. The results can be seen in the figures below with the corresponding time it took to compute them.



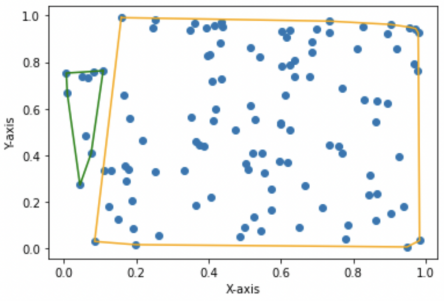
(a) Figure 4



(b) Figure 5



(c) Figure 6



(d) Figure 7

(a) In Figure 4, 15 points were computed in 0.1616 seconds.

(b) In Figure 5, 75 points were computed in 15.7626 seconds.

(c) In Figure 6, 100 points were computed in 23.09 seconds.

(d) In Figure 7, 115 points were computed in 68.9750 seconds.

4 Time Complexity

4.1 Number of Iterations

Let there be n points $\in \mathbb{R}^2$.
The algorithm loops through every possible combination of 2 dividing points for the convex hulls. There are n choose 2 different ways to select the 2 points for each iteration through the loop.
This can be represented as:

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)(n-3)...(2)(1)}{2(n-2)(n-3)(n-4)...(2)(1)}$$

Cancelling yields:

$$\frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

Therefore the algorithm will make $O(n^2)$ iterations through this loop.

For each iteration through this loop, the algorithm will check every line segment in each of the two convex hulls against each other.

4.2 Partition Runtime

The runtime of a given split is dependent not on n, but on the number of points on the convex hulls:
Let p denote the number of points on the convex hull left of or on the split line, and q denote the number of points on the convex hull right of the split line, so that $p + q = n$.
The runtime of any given split will be pq.
The worst possible run time for a split is the case when $p = \frac{1}{2}n$ and $q = \frac{1}{2}n$, so for simplicity I will just look at the cases where this is true: when all the points in the set are on the convex hull and the points are split equally between the convex hulls.

$$p + q = n \rightarrow pq = (n-q)q = (n-p)p$$

To get the run time in terms of n:

$$(n - \frac{1}{2}n)\ \frac{1}{2}n = (\frac{1}{2}n)^2 = \frac{n^2}{4} = pq = O(n^2)$$

Take a case where $p + q \neq n$, but instead some fraction of n, such as in Figure 8, where

$n = 15$ and $p = 4 = q$. Then:

$$p + q = 8 = \frac{8}{15}n,\ p = q = \frac{4}{15}n$$

.

$$pq = (\frac{8}{15}n - \frac{4}{15}n)\ \frac{4}{15}n = (\frac{4}{15}n)\ \frac{4}{15}n = \frac{16}{15^2}n^2 = 16$$

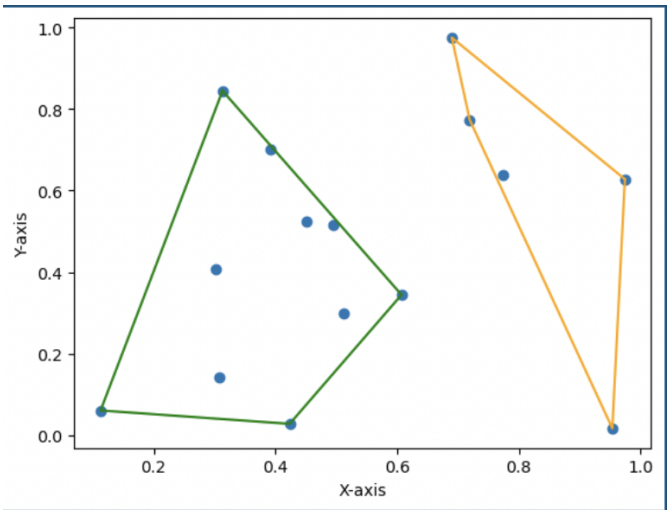


Figure 8: An average case

The runtime in terms of n can be analyzed using the ratio of combined boundary points on the convex hull to n, so let:

$\sigma = \frac{(p+q)}{n}$, σ is the ratio of boundary points on the convex hulls to points in the set.

Therefore:

$$p + q = \sigma n,\ \text{so:}\ pq = (\sigma n - q)q = \sigma nq - q^2$$

Let's assume the worst case, that:

$$q = \frac{1}{2}(p+q) = \frac{1}{2}\sigma n,\ \text{so:}\ \sigma nq - q^2 = \frac{1}{2}\sigma^2 n^2 - \frac{1}{4}\sigma^2\sigma^2 = \frac{1}{4}\sigma^2 n^2 = pq$$

The highest time complexity partitions maximize the function $\frac{1}{4}\sigma^2$. σ is a ratio, it is a decimal value between 0 and 1, so it's maximum value is 1. When $\sigma = 1$:

$$\frac{1}{4}\sigma^2 n^2 = \frac{1}{4}n^2, \text{which is the worst possible runtime for a partition.}$$

In Figure 9, every point in the set is on one of the convex hulls, so $\sigma = 1$ and

$$\frac{1}{4}n^2 = \frac{1}{4}144 = 36 \text{ comparisons are made.}$$

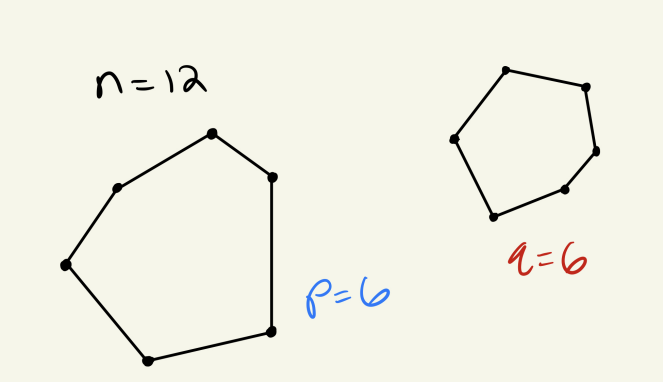


Figure 9: *Worst Possible Case (IPad Drawing)*

Over hundreds of observations of different splits using Python, in general I found that σ is usually approximately equal to $\sqrt{\frac{1}{n}}$ so that

$$\sigma^2 = \frac{1}{n}, \text{ and then: } \frac{1}{4}\sigma^2 n^2 \approx n$$

4.3 Overall Runtime

Therefore, for most average cases of iterations through the loop, the algorithm will find the distance between convex hulls in roughly $O(n)$ time. This means the total runtime of the entire algorithm is generally:

$$O(n^2) * O(n) = O(n^3)$$

However, there are very rare certain cases where most of the partitions will be computed in $O(n^2)$ time. This pushes worst case time complexity to:

$$O(n^2) * O(n^2) = O(n^4)$$

It is also to keep in mind that this is only for splits where:

$p = \frac{1}{2}(p+q)$ and $q = \frac{1}{2}(p+q)$, which is the highest complexity distribution of p+q.

If the split of points between the left convex hull and right convex hull were different, the results would be similar except the $\frac{1}{4}$ in:

$\frac{1}{4}\sigma^2 n^2$ would be a different fraction.

In conclusion the algorithm will almost always run in $O(n^3)$ time.

5 Applications

The furthest hyperplane problem is one that has many applications and uses, ranging from things like robotics and autonomous vehicles to biomedical applications, it can benefit a wide range of fields. The idea of maximizing the minimum distance between two hulls is something that can help to differentiate groups and ensure an efficient and effective setup. In this sense, this problem can be used for tasks like resource allocation and video game design.

5.1 General Applications

By using a convex hull algorithm, an autonomous vehicle could be told how to move from one point to another, avoiding any obstacles in the path. This algorithm would use hyperplanes and convex hulls to create 'neighborhoods' of points and store these in order to calculate the optimal path [5]. By maximizing the hyperplane in this, the vehicle would be given the most space and is the furthest away from any possible collisions or obstacles.

The use of hyperplanes is very important in biomedical imaging, when analyzing CT scans or MRIs, hyperplanes can be used to separate healthy tissue from pathological tissue, helping to assess the problem and further treatment. In addition, a convex hull could be used to help to classify an affected organ region and interpret an image or scan. In addition, classifying sequences of proteins and genes for patients can be hugely beneficial for detecting problems such as cancer [5].

When creating a video game, particularly a single player story game, the creation of the playable area can be benefitted by the use of the furthest hyperplane problem. This can ensure a good spread of locations, materials, and playable areas, to make the gameplay flow naturally and in an effective manner. Similarly, this algorithm can be applied to a business for resource allocation, to make production more efficient. By maximizing the minimum distance between resources, a company can ensure that they are spread effectively and in such a way that the production is not hindered by layout. By making sure there is a good and effective spread of resources, a company can be sure that they are not neglecting some areas while others are overstocked, ensuring that efficiency is not lost in that regard.

5.2 Support Vector Machines

The objective of a support vector machine algorithm is to find a hyperplane in N-dimensional space that will classify the data points. In particular, SVMs divide the datasets into classes, identifying a hyperplane that is equidistant from the two classes. Then test every possible hyperplane to find the maximum distance between the hyper-

plane and the two classes, in other words find the best separation of the data. There could be many different possible hyperplanes, but the objective is to find the plane with the maximum margin, classifying the data points into classes [3].

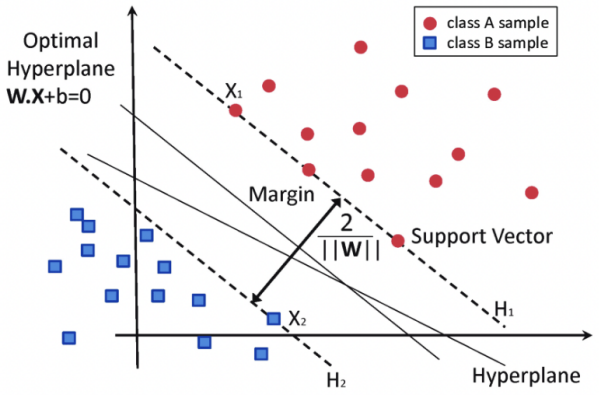


Figure 10: *Depiction of a hyperplane classifying a set of data points*

Support vector machines are used in a wide range of fields like face detection, which we use every day on our phones, classifying parts of the picture as face and not face to create a convex hull around the face. It is also used in handwriting recognition, assigning the pixels from scans of handwritings to vectors which can then be classified to help recognize a wide range of handwriting [11]. SVMs are also used in bioinformatics, for protein classifications, and identifying the classifications of genes and other biological problems [11]. Support vector machines are definitely the most commonly seen application of the furthest hyperplane problem. Because of their lack of specificity, SVMs can be applied to essentially any set of data in order to classify it effectively. This results in the wide range of applications for SVMs mentioned above, as well as many more possibilities.

6 Conclusion

In conclusion, I achieved my goal of creating an algorithm that computes the partition of a given point set that maximizes the distance between convex hulls. The main challenge was how to most efficiently calculate the minimum distance between two convex hulls. I employed a python based line segment comparison algorithm. The overall runtime of the algorithm was $O(n^3)$ on average. I found that it could run relatively quickly on Python for point sets with 100 points or less. However, as I increased this number to even just a couple hundred points, the time it took to return increased to multiple hours. There are optimizations that could be implemented to make the algorithm more efficient such as using hyperplanes, which has applications in robotic path planning and support vector machines.

Works Cited

[1] Berg, Mark de. Computational Geometry: Algorithms and Applications. Edited by Mark de Berg, 3rd ed., Berlin, Springer, 2008.

[2] DW, and Strahinja. "Minimum Distance between Two Convex Hulls Maximized." Computer Science Stack Exchange, 21 Aug. 2018, cs.stackexchange.com/questions/96430/minimum-distance-between-two-convex-hulls-maximized.

[3] Jain, Apurv. "Support Vector Machines(s.v.m)-Hyperplane and Margins." Medium, Medium, 25 Sept. 2020, medium.com/@apurvjain37/support-vector-machines-s-v-m-hyperplane-and-margins-ee2f083381b4.

[4] Liu, Bin, et al. "A Discriminative Method for Protein Remote Homology Detection and Fold Recognition Combining Top-n-Grams and Latent Semantic Analysis - BMC Bioinformatics." BioMed Central, BioMed Central, 1 Dec. 2008, bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-510.

[5] Meeran, S., and A. Shafie. "Optimum Path Planning Using Convex Hull and Local Search Heuristic Algorithms." Science Direct, 1997, https://www.sciencedirect.com/science/article/pii/S0957415897000330?ref=cra_js_challenge&fr=RR-1.

[6] "NumPy: The Absolute Basics for Beginners." NumPy, 2023, https://numpy.org/devdocs/user/absolute_beginners.html.

[7] Oliphant, Travis E. Guide to NumPy. Travis E. Oliphant, 2006.

[8] "Qhull manual." Qhull, 2023, http://www.qhull.org/html/index.htm#description.

[9] "Quick start guide — Matplotlib 3.8.2 documentation." Matplotlib, Matplotlib Develpement Team, 2012, https://matplotlib.org/stable/users/explain/quick_start.html#quick-start. Accessed 16 November 2023.

[10] SciPy Community. SciPy User Guide. SciPy Community, 2023. SciPy, https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide. Accessed 18 November 2023.

[11] Team, DataFlair. "Real-Life Applications of SVM (Support Vector Machines)." DataFlair, 8 Mar. 2021, data-flair.training/blogs/applications-of-svm/.

[12] The SciPy Community. "SciPy.Spatial.ConvexHull." SciPy, 2023, https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.ConvexHull.html. Accessed 24 November 2023.