The Nearest Neighbour Problem - A Survey

David Worley

For COMP 4750

Contents

1 Introduction

Which coffee shop is the closest so you can grab a cup without being late to class? Which gas station is closest so you minimize your risk of running out of fuel? Which post office is closest to you so you know where is best to drop off your mail? These are examples of problems people face every day, all oriented around the question, "If I'm here, whats the closest thing to me so I can get there fastest?" This question is simply an informal statement of the famous Nearest Neighbour Problem, also frequently called the Post Office Problem.

1.1 Problem Statement

Mathematically, the Nearest Neighbour Problem asks the following:

Given a query point q, and a set of points S, what is the point $p \in S$ such that $d(p,q) < d(r,q)$, $\forall r \in S, r \neq p$, for some distance function d.

The most clear example of the problem would be to use points in the plane, this would be the same as above with $S \subset \mathbb{R}^2$ and $d(p,q) = ||p \cdot q||$. for $p, q \in S$.

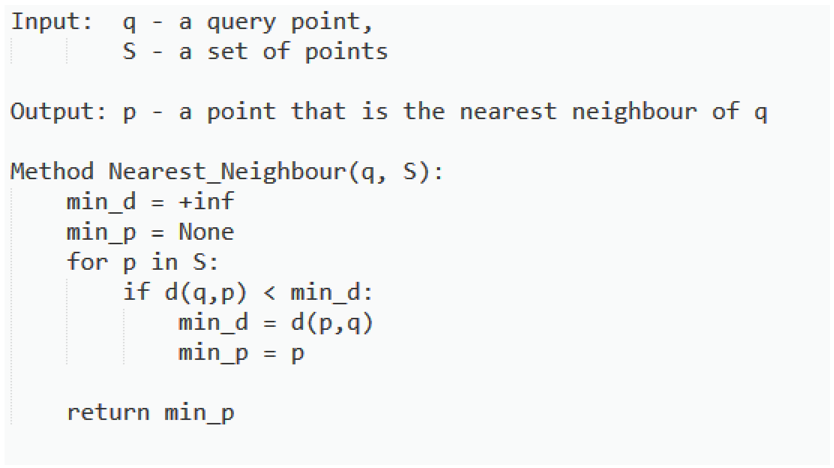
While this may be more clear, the problem naturally extends to any metric space due to its general statement, and in real world data it is far more common to have data with many more than two coordinates. Typically, the number of attributes each object of S has is referred to as the *dimensionality of the data*. For example, in $\mathbb{R}^{10}$, the dimensionality of the data would be 10.

2 Methodology

The nearest neighbour problem can be approached in numerous ways, and the best approach is not always obvious. In cases with high dimensionality or a very large amount of data, it may be better to find a "close enough" nearest neighbour in a fraction of the time than it is to find the exact nearest neighbour, in this scenario an approximate nearest neighbour approach can be employed. In other cases the exact nearest neighbour may be required, with an approximation simply not being good enough.

2.1 Brute Force Approach

The simplest method of solving the nearest neighbour problem is to check the distance from the query point to *all* other points of S, keeping track of the minimum at each check. This is the slowest approach, but on small sets of data it may be sufficient and desired due to its simplicity. This brute force approach takes $O(n)$ time, where n is the size of the data set S, but requires no additional space. This $O(n)$ time can become slow and lags behind other approaches as the size of the set S increases.[1]



```
Input:  q - a query point,
        S - a set of points

Output: p - a point that is the nearest neighbour of q

Method Nearest_Neighbour(q, S):
    min_d = +inf
    min_p = None
    for p in S:
        if d(q,p) < min_d:
            min_d = d(p,q)
            min_p = p

    return min_p
```

Figure 1: Psuedocode for a brute force nearest neighbour search

2.2 Space Partitioning Methods

A faster method for solving the nearest neighbour problem is to employ the use of trees to quickly partition the data set, limiting the number of points one needs to check to find the nearest neighbour. A common structure for this approach is a kd-tree.

In a kd-tree, each internal node consists of a splitting plane that divides the search region in half. Under this construction, a nearest neighbour search can be performed in two parts, first by searching for the query point in the tree and following the path to the leaf. If the path has k nodes, then the search space has been split in half k times. Then this path can be unravelled to search for candidate points close to the splitting planes by considering a hypershpere with radius equal to the current minimum distance, and continuing down the other side of the splitting plane if the sphere intersects the splitting plane to search for closer candidate points.

For a set S of size n, the kd-tree requires $O(n)$ space and can execute a nearest neighbour query in, on average, $O(logn)$ time, making this approach significantly faster on average than the brute force approach.[2] However, for randomly distributed points the worst case of the search is $O(kn^{1-\frac{1}{k}})$, where k is the dimensionality of the data. in this case, the increase over the brute force method is marginal. [3]

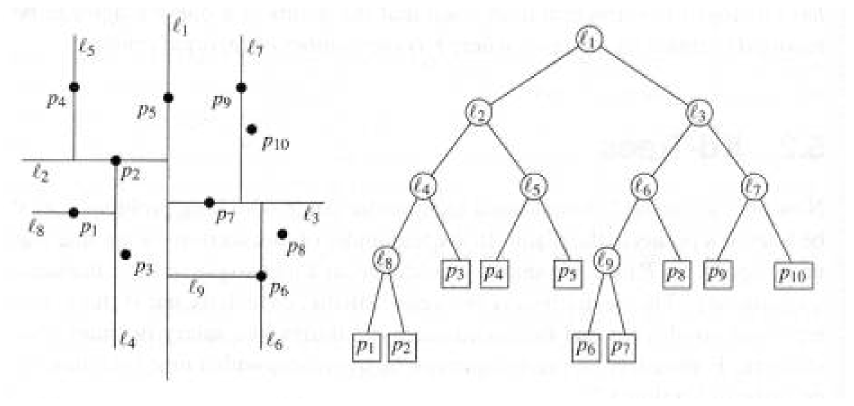


Figure 2: An example of a point set S and its corresponding k-d tree

2.3 Greedy Approximation

The methods mentioned previously are both exact methods, guaranteed to give the nearest neighbour to the query point. However, there are cases where an approximate nearest neighbour is essentially just as good. One example could be that it does not make a large difference if you go to a coffee shop 310 meters when the closest one is 300 meters away, as the difference is largely negligible and you still get your coffee. In cases like this the use of an approximate nearest neighbour algorithm can be employed. An important quality for approximate nearest neighbour is that the approximation is, of course, good. Because of this many greedy methods are designed such that the distance they return will be less than a constant multiple times the true minimum distance [4]

The current best known approximation methods for the nearest neighbour approaches are greedy in nature, with a common approach making use of proximity graphs created using elements of S as vertices, with an edge between two vertices if they are within a certain distance from each other. A nearest neighbour approximation for a query point q can then be found by picking a vertex v of the proximity graph G, and calculating the distance between v and q, as well as the distance between q and all neighbours of v in G. If a neighbour of v is closer to q, than the search moves to that vertex and repeats the process, only terminating when the search arrives at a vertex whose neighbours are all further from the query point than it is.[5]

Another approach is to convert the exact kd-tree approach into an approximation method. This can be done by simply setting an upper bound on the number of points checked and returning the minimum after either the search has been exhausted or the upper bound is complete.

2.4 Dimensionality of Data

The dimensionality of the data is an important aspect for nearest neighbour problems, as data with higher dimensionality is better suited to different approaches than low dimensionality data. A common issue faced by high dimensional data is informally known as the *Curse of Dimensionality*, and refers to the tendency for nearest neighbour solutions to degrade in performance on data with high dimensionality. For example, in the kd-tree scenario outlined above it is common for higher dimensional data to be more sparse, causing the algorithm to need to check in a wider range around its splitting plane, frequently causing it to search down many paths. This can be so extreme that the query runs in nearly linear time, erasing the speed benefit of the splitting approach while still using $O(n)$ space. [6]

It is important to note, however, that this is not always the case. The above issue typically occurs when some of the data attributes are less impactful in the distance calculations (and can be considered "noise" data, instead of valuable information). Thus, it is very important to only include necessary data and to apply dimensionality reduction on data when possible to alleviate this issue.

3 Some Common Variations

The Nearest Neighbour Problem has many variations, each with their own complexities, intricacies, and approaches.

3.1 k-Nearest Neighbours

The k-nearest neighbours problem is a very similar problem to the classic nearest neighbour problem. Instead of asking for the single closest neighbour, to the query point q, it asks for the k nearest points. This variation frequently occurs in the fields of statistics and machine learning due to its applications in regression and classification which will be detailed more thoroughly in Section 5.3.

This variation can be solved using brute force in $O(k + nlogn)$ by checking the distance from the query point q to every point of S and storing it in an array, then sorting the array and taking the first k elements.

Another approach is to use a kd-tree or a ball-tree, which allows for query times of $O(klogn)$.[7] In this approach, the k-d tree is constructed as normal, and can then be queried k times, removing the nearest neighbour each time. This still can be improved, as the k points found will be relatively close to each other. Keeping this in mind, its possible to modify the k-d tree algorithm to find all k nearest neighbours in one search. [8]

3.2 All Nearest Neighbours

The All Nearest Neighbours problem is equivalent to asking what the nearest neighbour to q is, for every q in S. This can clearly be solved by running a nearest neighbour query on each point of S (and not considering the $d(q, q)$ case), but this approach is very slow and makes many repeated calculations.

A better approach would be to exploit the repeated calculations (for example, the distance from x to y is the same as the distance from y to x) to reduce the number of distance calculations needed. Its been proven that $O(nlogn)$ is the optimal time complexity for an all nearest neighbours algorithm. A notable example by P. Vaidya achieves this by splitting points into smaller and smaller boxes, with each box containing some neighbourhood information. Vaidya also proved that his algorithm is optimal up to a constant factor. [9]

3.3 Fixed Radius Nearest Neighbours

The fixed-radius nearest neighbour search is slightly different as it is the only variation mentioned here where the number of neighbours being searched for is not known beforehand. The fixed radius nearest neighbour problem asks, "For a radius r, and a query point q, which points p of S have $d(p, q) < r$?" The problem can be equivalently rephrased as, "Given an n-dimensional hypersphere of radius r, centred at a given point q, which points p of S are contained within the hypersphere?", where the dimensionality of the data is n.

The problem can also be solved with kd-trees by visiting a nodes subtree if and only if the subtree overlaps with the hypersphere of radius r, centered at the query point.

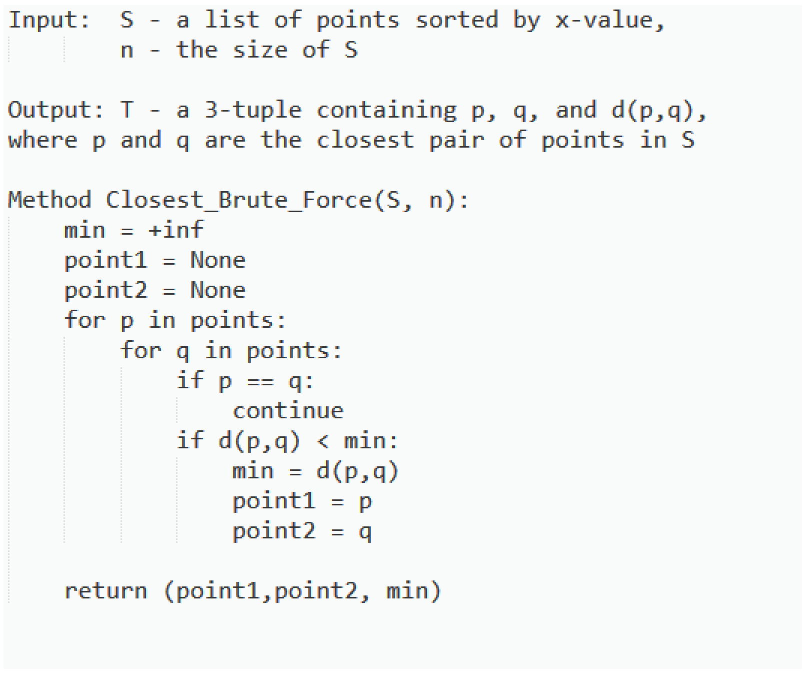
4 The Closest Pair of Points Problem

The Closest Pair of Points problem asks the question, "Given a set of points S, find the two points, p and q, of S such that $d(p, q)$ is minimal."

4.1 The Brute Force Solution

The problem can be solved in $O(n^2)$ time by calculating the distance from each point to every other point, tracking the minimum at each comparison. This process takes no extra space, but is very slow as the size of S increases.

The above psuedocode demonstrates the simplicity of the brute force approach.



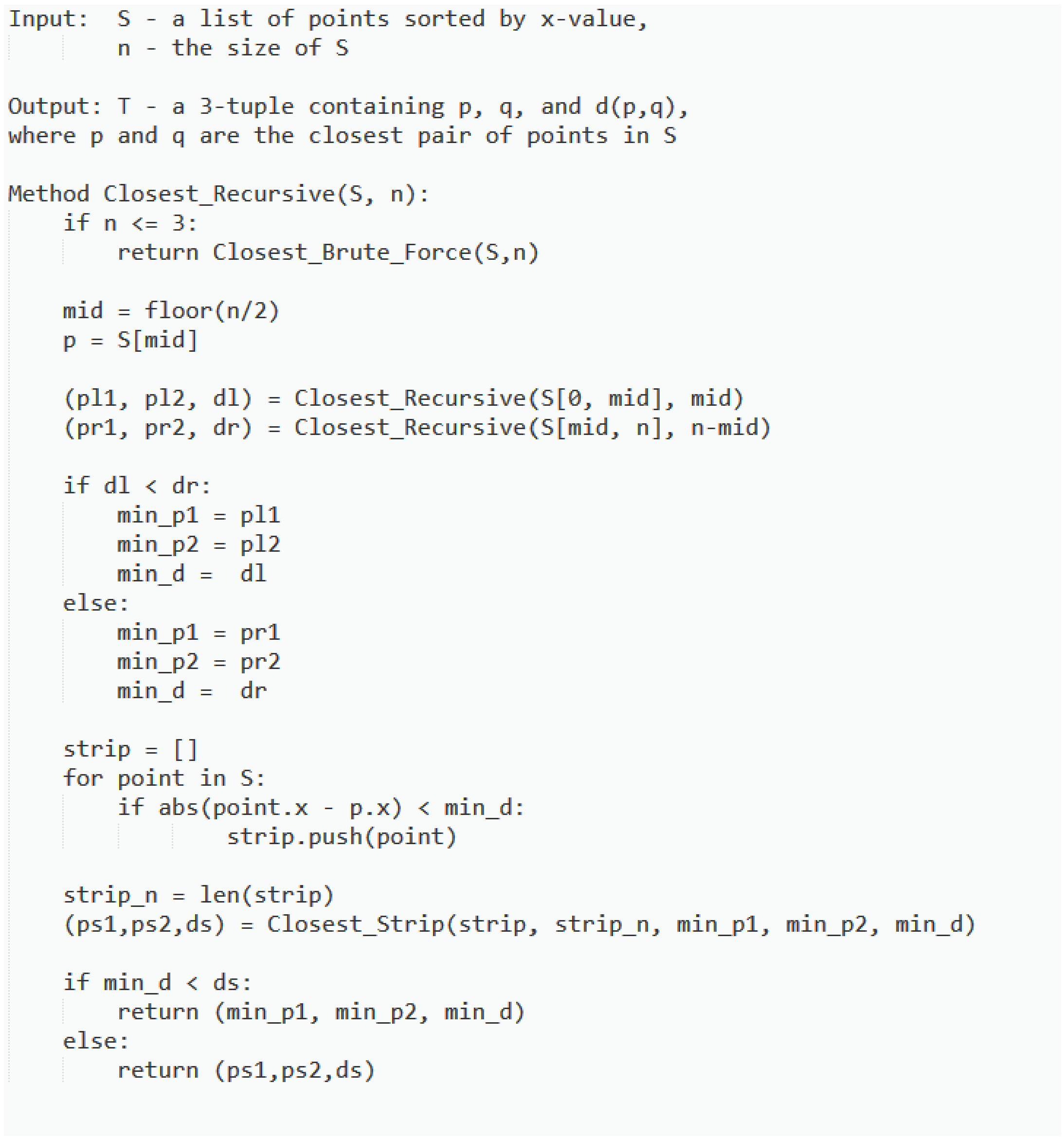
```
Input:  S - a list of points sorted by x-value,
        n - the size of S

Output: T - a 3-tuple containing p, q, and d(p,q),
where p and q are the closest pair of points in S

Method Closest_Brute_Force(S, n):
    min = +inf
    point1 = None
    point2 = None
    for p in points:
        for q in points:
            if p == q:
                continue
            if d(p,q) < min:
                min = d(p,q)
                point1 = p
                point2 = q

    return (point1,point2, min)
```

Figure 3: Psuedocode for the brute force closest pair solution

4.2 A Divide and Conquer Approach

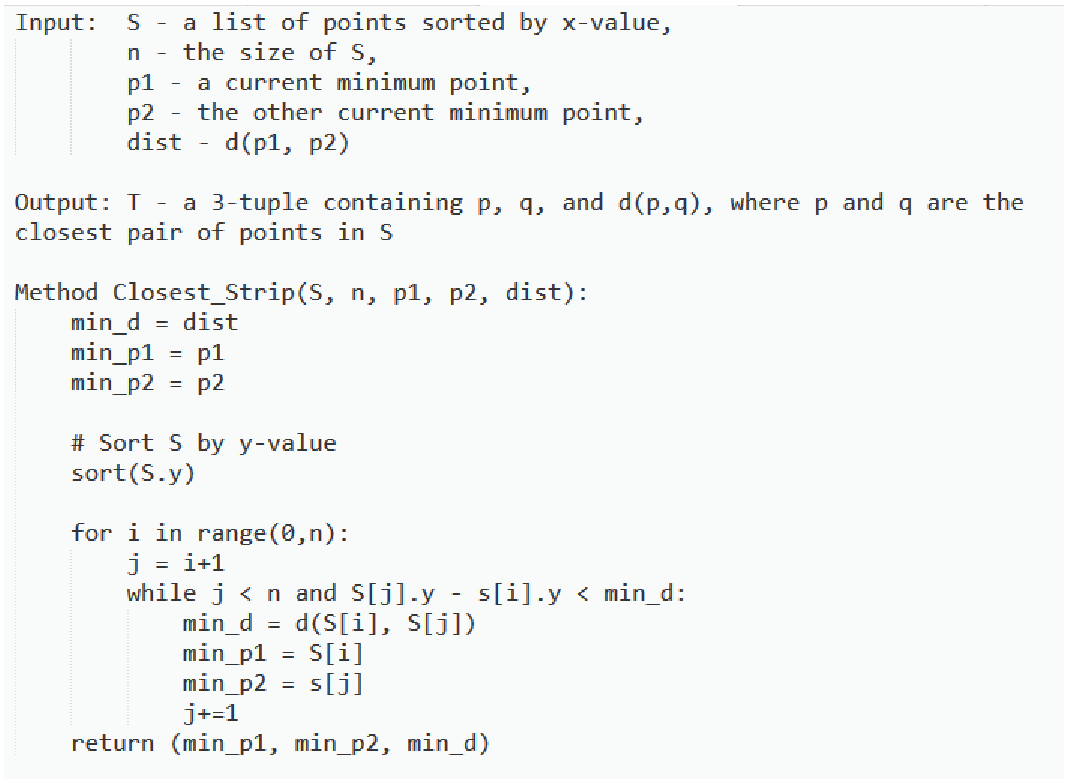
The problem can also be solved by employing a recursive divide and conquer technique on a point set sorted by x-value such that it divides the set into a left side, a right side, and a middle strip at each division. The closest pair on the left and right side as well as in the middle strip are calculated, and the minimum of those is kept going into the next recursive call. The base case for the recursion is when the split sides have 3 or fewer points, in which case the brute force algorithm is run on them to calculate the closest pair.

This approach results in a query time of $O(nlogn)$, and since all operations are done on the original list of points, no additional space is required. The main drawbacks of this approach is that it does not scale with the dimensionality of the data, and is only practical in the planar case (dimensionality equal to two). Despite this, its speed and simplicity make it one of the best approaches for the planar closest pair problem.



```
Input:  S - a list of points sorted by x-value,
        n - the size of S

Output: T - a 3-tuple containing p, q, and d(p,q),
where p and q are the closest pair of points in S

Method Closest_Recursive(S, n):
    if n <= 3:
        return Closest_Brute_Force(S,n)

    mid = floor(n/2)
    p = S[mid]

    (pl1, pl2, dl) = Closest_Recursive(S[0, mid], mid)
    (pr1, pr2, dr) = Closest_Recursive(S[mid, n], n-mid)

    if dl < dr:
        min_p1 = pl1
        min_p2 = pl2
        min_d =  dl
    else:
        min_p1 = pr1
        min_p2 = pr2
        min_d =  dr

    strip = []
    for point in S:
        if abs(point.x - p.x) < min_d:
                strip.push(point)

    strip_n = len(strip)
    (ps1,ps2,ds) = Closest_Strip(strip, strip_n, min_p1, min_p2, min_d)

    if min_d < ds:
        return (min_p1, min_p2, min_d)
    else:
        return (ps1,ps2,ds)
```

Figure 4: Psuedocode for the divide and conquer closest pair solution



```
Input:  S - a list of points sorted by x-value,
        n - the size of S,
        p1 - a current minimum point,
        p2 - the other current minimum point,
        dist - d(p1, p2)

Output: T - a 3-tuple containing p, q, and d(p,q), where p and q are the
closest pair of points in S

Method Closest_Strip(S, n, p1, p2, dist):
    min_d = dist
    min_p1 = p1
    min_p2 = p2

    # Sort S by y-value
    sort(S.y)

    for i in range(0,n):
        j = i+1
        while j < n and S[j].y - s[i].y < min_d:
            min_d = d(S[i], S[j])
            min_p1 = S[i]
            min_p2 = s[j]
            j+=1
    return (min_p1, min_p2, min_d)
```

Figure 5: Psuedocode for finding the closest pair within a middle strip

One interesting observation is that the calculation of the closest pair within the strip (fig. 5) appears to be $O(n^2)$ at a glance, however, for each point in the inner loop, there are only 6 points that need to be checked[10], making the complexity of the strip method $O(n)$.

This can be seen by considering a rectangle of side length d and $2d$, it can be shown by placing circles of radius d on each corner of the rectangle, as well as at both halfway points of the longer line, that this box can only contain at most 6 points. This observation means we only need to check 6 points for each point in S, and thus, the Closest_Strip algorithm runs in $O(nlogn)$ time as well (due to the sorting of S).

Analysis on n=50,000 unique points in $\mathbb{R}^2$ with coordinates in $[-5000, 5000]$ showed the divide and conquer algorithm running approximately 275x quicker than the brute force algorithm, based off of the average of 10 random trials on algorithms implemented in Rust.

4.3 Relation to Voronoi Diagram

Recall that the Voronoi region of a point p is the region surrounding the point such that all points within the region are closer to p than any other point, and the Voronoi Diagram for a point set is the collection of Vonoroi regions for each point in the set. Its clear that the closest pair of points will have adjacent Voronoi regions, and thus the closest pair can be found by checking the distance between points with adjacent Voronoi regions and storing the minimum.

The Delaunay Triangulation relates similarly to the closest pair problem. In the Delaunay Triangulation, the dual of the Voronoi Diagram, the closest pair of points corresponds to an edge, so each edge can be checked to find the closest pair.

Both the Voronoi diagram and the Delaunay Triangulation can be calculated in $O(nlogn)$ time and, once either is obtained, the closest pair of points can be found in $O(n)$ time using the above observations.[10]

5 Applications

Due to its generality and importance, the nearest neighbour problem arises in many different fields to solve problems. Nearest neighbour approaches can be useful for any problem in which an objects similarity to other objects in a set or database is important.

5.1 In DNA Sequencing

One fascinating example of the nearest neighbour problem's applicability is in DNA sequencing. Given a query DNA sample, finding its nearest neighbours in a set of DNA samples with known classifications (proneness to disease for example), can give approximations of what the classification of the query DNA is as well. This is particularly useful on large data sets where the k-nearest neighbours to a sample are more likely to accurately reflect the classification of the sample, as too sparse of a dataset may give results that are largely meaningless or innaccurate.

5.2 Plagiarism Detection

Another interesting example of an application of the nearest neighbour problem is in plagiarism detection. For each submitted assignment, how close it is to its nearest neighbour in a database of entries would be an accurate measure of how much identical content is shared between the two documents. There are certainly more complexities in the distance calculations for these objects, with a modified Levenchstein distance being a likely candidate for measuring similarity by measuring the amount of operations (character removal, changing, or a cyclic shift of the string) it takes to transform one string of characters into another.

5.3 Machine Learning

Nearest neighbour problems are frequently employed in classification and regression, with the k-nearest neighbour algorithm being a very fundamental algorithm in classification. Consider a query picture and a set of pictures whose classification is known, then the classification of the query picture can be assumed based off of the classification of its k-nearest neighbours. Similarly, if you had a set of points, and their corresponding values, you could find the approximate value of a query point q by finding the k-nearest neighbours, and taking the average of their values, this allows k-nearest neighbours to be used to model regression when enough test data is obtained.

One large drawback of this method is that if the data is dominated by points of one type of classification, its likely that the k-nearest neighbours for a query point will include points of that classification, regardless on whether its accurate or not. This makes data reduction and proper data collection extremely important when it comes to using k-nearest neighbours for classification problems. For example, if a database has 20,000 pictures of cats, but only 5,000 pictures of dogs, then a collection of k-nearest neighbours for a query picture of a dog may be more likely to include pictures of cats that skew the classification and accuracy of the algorithm.

Conveniently enough, a modified nearest neighbour search known as Condensed Nearest Neighbour search can be used to reduce datasets. CNN search uses a set of prototype data (known to give accurate results) and, for each prototype, selects its nearest neighbours with different classifications to create a new, smaller dataset that can more reliably be used for classification.

6 Current Research Focuses

Due to the large amount of focus on machine learning and artificial intelligence, current research and applications regarding the nearest neighbour problem are skewed towards these results, with a large portion of papers being related to applications of k-nearest neighbour to machine learning. This is unsurprising since the k-nearest neighbours algorithm is one of the most popular classification approaches in machine learning due to its speed and simplicity.

Applying nearest neighbour search to new problems and areas like animal behaviour is also a recurring research theme when looking at recent nearest neighbour publications.

This overall trend shows that as of right now the research climate around this problem is applying nearest neighbour results to other fields and areas, with a smaller focus on improving the nearest neighbour approaches themselves. This could largely be due to multiple variations having near optimal algorithms already, such as all nearest neighbour and k-nearest neighbour. However, due to its prevalence in artificial intelligence, variations of k-nearest neighbour algorithms have been created for solving classification problems. [11]

While the fundamental algorithms are near optimal, plenty of work has been done in improving specialized nearest neighbour algorithms in different applications. For example, an improved nearest neighbour algorithm for DNA clustering was recently developed. [12]

7 Future Work

Potential areas for future study would be to examine more specialized variants of the nearest neighbour problem, such as the ones used in DNA sequencing and a closer look into k-nearest neighbour algorithms used in classification. Another interesting area of future work would be to provide more implementation and runtime details for nearest neighbour algorithms and variations, such as a kd-tree implementation or proximity graph based approximation method. These could be compared in runtime and accuracy to a brute force implementation on separate data sets to weigh the benefits of the separate approaches and highlight the worst case scenarios for both.

8 Conclusion

It is evident from its wide range of variations and applications that the nearest neighbour problem is an important, fundamental problem that spans many different fields in terms of its applicability. Through an analysis of the closest point variation, we can see the extreme run-time differences that approaching the nearest neighbour problem in a smart manner can create. It can also be seen that its applications to areas like machine learning reinforce its importance, as it allows for greater progress to develop in other areas. It seems likely that as the nearest neighbour problem is applied to more and more fields, more variations will arise to efficiently solve these specialized problems.

9 References

[1] Weber, Roger; Schek, Hans-J.; Blott, Stephen (1998). *"A quantitative analysis and performance study for similarity search methods in high dimensional spaces"* (PDF). VLDB '98 Proceedings of the 24rd International Conference on Very Large Data Bases: 194–205.

[2] Andrew Moore. *"An introductory tutorial on KD trees"* (PDF). Archived from the original (PDF) on 2016-03-03. Retrieved 2008-10-03.

[3] Lee, D. T.; Wong, C. K. (1977). *"Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees"* Acta Informatica. 9 (1): 23–29. doi:10.1007/BF00263763.

[4] Andoni, A.; Indyk, P. (2006-10-01). *"Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions."* 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06). pp. 459–468.

[5] Malkov, Yury; Yashunin, Dmitry (2016). *"Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs"*. arXiv:1603.09320

[6] Glen, S. *"Dimensionality & High Dimensional Data: Definition, Examples, Curse of"* From StatisticsHowTo.com: Elementary Statistics for the rest of us! https://www.statisticshowto.com/dimensionality/

[7] Adamczyk, J. (2020, September 14). *"K nearest neighbors computational complexity."* Retrieved November 29, 2020, from https://towardsdatascience.com/k-nearest-neighbors-computational-complexity-502d2c440d5

[8] Kakde, H. (2005, August 25). *"Range Searching Using Kd Tree"* Retrieved December 10th, 2020, from http://www.cs.utah.edu/ lifeifei/cis5930/kdtree.pdf

[9] Vaidya, P. M. (1989). *"An O(n log n) Algorithm for the All-Nearest-Neighbors Problem"*. Discrete and Computational Geometry. 4 (1): 101–115.

[10] M. I. Shamos and D. Hoey, *"Closest-point problems"* 16th Annual Symposium on Foundations of Computer Science (sfcs 1975), USA, 1975, pp. 151-162, doi: 10.1109/SFCS.1975.8.

[11] Ullah, Rafi, Ayaz H. Khan, and S. M. Emaduddin. *"ck-NN: A Clustered k-Nearest Neighbours Approach for Large-Scale Classification."* (2019).

[12] Chen, G. "An improved nearest neighbour algorithm for DNA sequence clustering" Nanyang Technological University, 2020