Acceleration of Binning Nearest Neighbour Methods

Michael Greenspan[†] Guy Godin[†] Jimmy Talbot[‡]

[†]Visual Information Technology Group
Institute for Information Technology
National Research Council Canada

[‡]University of Toronto
and MD Robotics Ltd.

corresponding author : Michael.Greenspan@nrc.ca

keywords : *nearest neighbour problem, closest point, computational geometry*

Abstract

A new solution method to the Nearest Neighbour Problem is presented. The method is based upon the triangle inequality and works well for small point sets, where traditional solutions are particularly ineffective.

Its performance is characterized experimentally and compared with k-d tree and Elias approaches. A hybrid approach is proposed wherein the triangle inequality method is applied to the k-d tree and Elias bin sets. The hybridization is shown to accelerate the k-d tree for large point sets, resulting in ~ 20% improvement in time performance. The space efficiencies for both the k-d tree and Elias methods also improve under the hybrid scheme.

1 Introduction

Let $\mathbf{P} = \{\vec{p}_i\}_1^n$ be a set of n points in a k-dimensional space, and let $\vec{q}$ be a query point. A statement of the *Nearest Neighbour Problem* is;

Find the point $\vec{p}_c$ in **P** *which is the minimum distance from $\vec{q}$, i.e.*

$$\|\vec{q} - \vec{p}_c\| \leq \|\vec{q} - \vec{p}_i\| \ \forall \ \vec{p}_i \in \mathbf{P} \quad (1)$$

This is a classical problem of computational geometry, and is encountered within computer vision and general pattern recognition. As an example, nearest neighbour determination is the rate determining step in Iterative Closest Point methods, which are ubiquitous within the field of range image processing [1].

We will assume that $\vec{p}_c$ is unique. We also assume that the distance metric is the Euclidean distance and denote it as $D_{ij} = \|\vec{p}_i - \vec{p}_j\|$. Note that the acceleration methods we propose apply to any metric that satisfies the triangle inequality. For convenience, we denote the distance of any $\vec{p}_i$ to the query point $\vec{q}$ with a single subscript, i.e. $D_i = \|\vec{p}_i - \vec{q}\|$.

The exhaustive approach is to calculate D_i for each $\vec{p}_i \in \mathbf{P}$ and identify the minimum, which has a computational complexity of $O(n)$. The object of any efficient solution is to identify $\vec{p}_c$ without explicitly calculating every D_i, with the cost of pruning less than that of the distance computations that it is removing.

The most common and most promising [2, p. 810] solutions are *k-d tree* [3] and *Elias* [4, 5] methods, both of which use binning. The k-d tree is a strictly binary tree in which each node represents a partition of the k-dimensional space. The root of the tree represents the entire space, and the leaf nodes represent subspaces containing mutually exclusive small subsets of **P**. At any node, only one of the k dimensions is used as a discriminator, or *key*, to partition the space. When the tree is traversed, the single scalar value of the key of $\vec{q}$ is compared with the node key value, and the appropriate branch is followed. When a leaf node is encountered, all of the points resident in the leaf's bin are tested against $\vec{q}$. Depending on the closeness of the match at a leaf node, and the boundaries of the space partition, the traversal may backtrack to explore alternate branches. The time complexity of traversing the k-d tree has been shown to be proportional to $\log n$ [6].

Whereas k-d tree methods partition space hierarchically and irregularly, a simple and general method for nearest neighbour search groups the data points into subsets that form regular (congruent and non-overlapping) subregions. Such an approach is often referred to as the Elias method [4, 5]. The nearest point to the query point $\vec{q}$ is found by searching the subregions in order of their proximity to $\vec{q}$, until the distance to any remaining region is greater than the distance to the nearest point found, or until all subregions have been processed. Points in each searched subregion are exhaustively examined. Simple modifications to the method allow the search for the K nearest neighbours, or for all

points closer than a threshold distance. The partitions are axis-aligned, forming squares in 2-D, cubes in 3-D, or hypercubes in k-D space for $k \geq 4$.

Other solution approaches have exploited the triangle inequality, including a branch and bound method [7], and an approach which aggregates the points in $\mathbf{P}$ into sets equidistant from some reference [8]. Vidal Ruiz [9] notes that there exists a class of nearest neighbor methods all of which use the triangle inequality as the main discriminator, and which differ primarily in their search strategies. In his AESA method [9, 10], the distances between all points in $\mathbf{P}$ are precomputed and stored. At runtime, an *anchor point* is randomly selected from $\mathbf{P}$, and its distance to $\vec{q}$ is calculated. Based upon the precomputed distances, any $\vec{p}_i \in \mathbf{P}$ which falls outside of the upper or lower bounds of the triangle inequality are disqualified. Until only a single point remains, a new anchor point is selected and the process iterates. Due to the expensive preprocessing and large memory requirements, the straightforward application of this method is impractical for all but small point sets.

Recently, Nene and Nayar [11] described a simple projection method which was efficient mainly in high dimensional spaces.

1.1 Performance Evaluation Metrics

One way to predict the expected relative performance of different methods is by comparing their complexity expressions, which are functions that relate the number of fundamental computations to the size of one or more input variables. For the nearest neighbour problem, the complexity expression almost always relates the number of distance computations to the size of $\mathbf{P}$. Unfortunately, these expressions do not tell the whole story, as multiplicative factors are not included. The multiplicative factors may have a large effect on the true run time expense, particularly at boundaries such as very large or very small inputs. Complexity expressions are also independent of implementation specifics, which may introduce large variations in run time expense.

In this work, we empirically compare the performance of a number of nearest neighbour methods based upon two metrics, E_c and E_t. For a given method $\mathcal{M}$, $\mathbf{P}$ of cardinality n, and $\vec{q}$, $E_c^{\mathcal{M}} = \frac{n}{m}$, where m is the number of points for which D was actually computed. Similarly, for a given implementation of $\mathcal{M}$, and an implementation $\mathcal{E}$ of the above described exhaustive method, $E_t^{\mathcal{M}} = \frac{t(\mathcal{E})}{t(\mathcal{M})}$, where $t()$ is the measured run time.

While E_c is a direct measure of computational complexity, E_t is dependant upon implementation specifics. Each implementation was executed on the same platform (SGI R8000 75 MHz) and compiled with the same compiler options, and a moderate and equivalent amount of attention was afforded to the optimization of each. For example, square root calculations were avoided wherever possible, but no assembly level coding or other machine specific techniques were used. It is believed that these implementations (and therefore the values of E_t) are representative of what a practitioner in the field would typically produce. It is also believed that these results would scale equivalently with any other implementation optimizations, so long as they were applied equally accross all methods.

2 K-d Tree

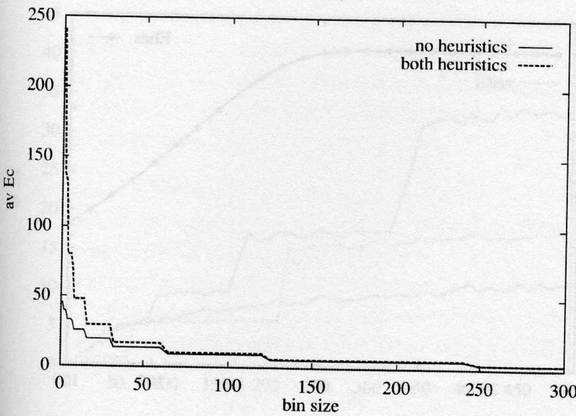
Friedman et. al [6] showed that the efficiency E_c of a k-d tree is optimal when B, the number of points per bin, is equal to 1. They also noted that the large amount of backtracking which occurs for small B comes at a time cost. The optimal value of E_t therefore occurs for a larger B.

Backtracking is controlled by two routines, Bounds_Within_Ball (**BWB**) and Bounds_Overlap_Ball (**BOB**). **BWB** is called when a leaf node is encountered during traversal, and a new current closest point $\vec{p}_i$ is identified. The sphere centered at $\vec{q}$ with radius D_i is compared to the boundary of the space partition represented by the leaf. If the sphere completely resides within the leaf partition, then no other closer point exists, and the search can terminate. Otherwise, the traversal must backtrack.

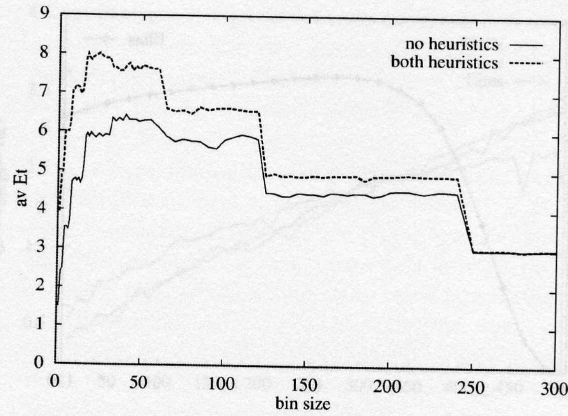
Similarly, the **BOB** routine determines whether it is necessary to investigate a particular branch. If the above described sphere does not overlap with a partition represented by a particular node, then none of the points in the subtree rooted at that node are closer than D_i to $\vec{q}$, and the traversal need not explore that branch.

To observe the effect of B on E_c and E_t, we calculated these values by varying B for a set of $n = 10^3$ uniformly randomly distributed points and a similar set of query points, the results of which are plotted in Figure 1. We ran the test with both the k-d tree and a modified version based on some additional heuristics, aimed at improving performance. These heuristics were derived from the observation that the smaller the node boundary, the more likely that the **BOB** test would fail, halting exploration of the subtree.

Instead of using the median of the discriminator key to determine which branch to explore, we pre-compute the actual boundaries of the point set comprising the node. This is useful to exploit the space that can exist between the adjacent boundaries of two sibling nodes. When using the discriminator to compute those boundaries, sibling nodes share a common boundary, even though the node which does not include the partitioning point may not contain points that are near the partition. This heuristic is also useful when the k-d tree is queried for all points within a certain radius of the query point; if the search radius is smaller than the distance of the query point to k-d tree, the search terminates immediately. The algorithm described in [6] sets the boundaries

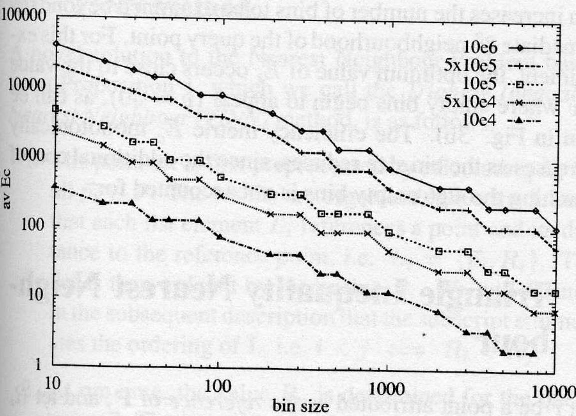


a) E_c vs. B

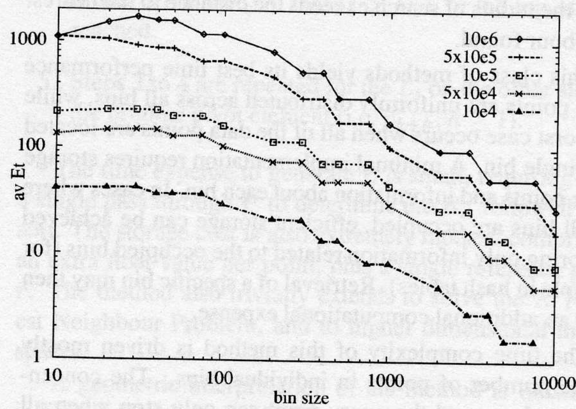


b) E_t vs. B

Figure 1: K-d Tree Performance vs. Bin Size, $10^4 \leq n \leq 10^6$



a) E_c vs. B



b) E_t vs. B

Figure 2: K-d Tree Performance vs. Bin Size, $10^4 \leq n \leq 10^6$

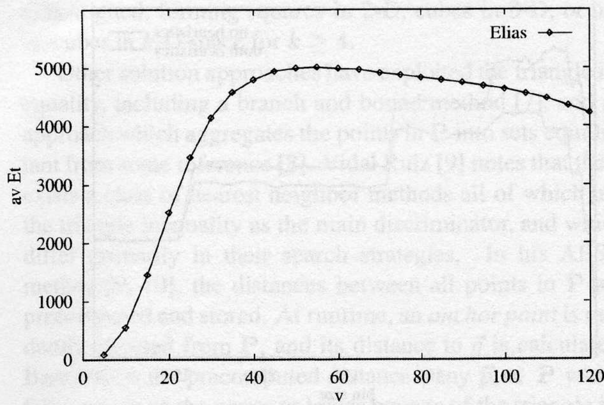
for the root node of a k-d tree at infinity, rendering this test unusable.

We also used the variance (as in [12]) instead of the spread (as in [3]) to select the discriminator. This favors child nodes whose boundaries are further apart, potentially creating a larger space between two sibling nodes. If the sphere of radius D_i is near the boundary of the current node that is adjacent to its sibling, then there is less likelihood that the **BOB** test will succeed.

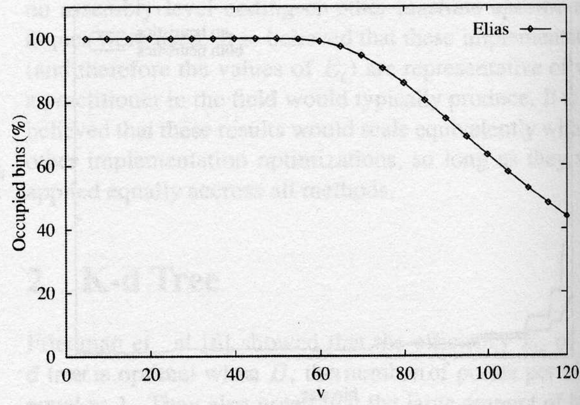
As illustrated in Figure 1, these heuristics appear to improve both the E_c and E_t efficiencies. It is also clear from 1b) that the value of E_t is optimized for $B \sim 20$. This experiment was repeated for point sets varying in size from $n = 10^4$ to 10^6. The results, plotted in Figure 2, show that the optimal value of E_t occurs at roughly the same value of B as n increases. This is not surprizing, as each subtree is emanating from an internal node is also a k-d tree, and the structure is likely to adhere to Bellman's *Principle of Optimality* [13].

3 Elias Method

This class of methods divides the space into congruent and non-overlapping sub-regions, or bins. Each bin contains a list of the points that fall within its boundaries. The closest point is searched first inside the bin where $\vec{q}$ is located. If the bin is empty, or if the distance between $\vec{q}$ and the closest point found is greater than the distance to any other bin, then the search is also performed on the list of points contained in that bin. This process is repeated until the distance to any unexplored bin is greater than the distance to the closest point found. In practice, the known spatial ordering of the bins is used to restrict the search: bins are accessed in a concentric order around the query point, and the search stops



a) E_t vs. v



b) % occupied bins vs. v

Figure 3: Elias Performance w.r.t. v

when the radius of search exceeds the distance to the nearest neighbour found.

This class of methods yields its best time performance when points are uniformly distributed across all bins, while the worst case occurs when all of the data points are located in a single bin. A minimal implementation requires storage of the points and information about each bin. In cases where not all bins are occupied, efficient storage can be achieved by storing only information related to the occupied bins (for example in hash tables). Retrieval of a specific bin may then incur an additional computational expense.

The time complexity of this method is driven mostly by the number of points in individual bins. The concentric search around the query point can only stop when all bins that are closer than the current closest point found have been explored. When all bins in the structure contain at least one point, then at most 3^k-2 (where k is the dimension of the space) bins adjacent to the bin which encloses the query point will need to be examined. If the proportion of empty bins is large, then the search radius will be increased. The actual cost of this operation depends on the efficiency of the implementation of the bin data structure as well as its access functions.

As an example, Fig. 3 shows the behaviour of an implementation for 3 dimensions, using 10^6 random points distributed over a cube. A large number (10^5) of random queries over the cube are performed and the values of E_t are measured. The abscissa variable v is the number of bins along each axis, therefore the total number of bins is v^3.

Two opposing phenomena influence the optimal time efficiency as a function of v. Larger bins contain more points, thus increasing the time expense of the exhaustive search in each bin. Conversely, beyond a critical point determined by the point distribution in space, a smaller bin size increases the proportion of empty bins for a given point set. This in turn increases the number of bins to be examined beyond the immediate 3^3 neighbourhood of the query point. For this experiment, the optimum value of E_t occurs close to the value of v where empty bins begin to appear ($v = 50$), as can be seen in Fig. 3b). The efficiency metric E_c monotonically increases as the bin size reduces, since the additional cost of searching through empty bins is not accounted for.

4 Triangle Inequality Nearest Neighbour

Let $\vec{r}$ be a point attributed as the *reference* of $\mathbf{P}$, and let R_i denote the distance of $\vec{p}_i$ to $\vec{r}$, i.e. $R_i = \|\vec{p}_i - \vec{r}\|$. We may set $\vec{r}$ to be any point relative to $\mathbf{P}$, including some $\vec{p}_i \in \mathbf{P}$.

For any two points $\vec{p}_i$ and $\vec{p}_j$ a statement of the *triangle inequality* is;

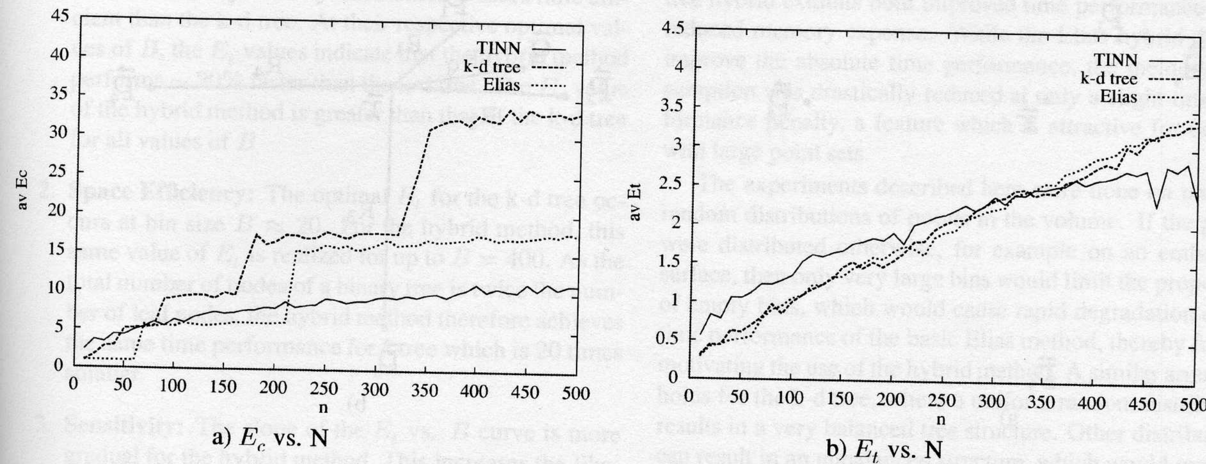
$$|R_j - R_i| \leq D_{ji} \leq R_j + R_i \tag{2}$$

The equality limit of the lower bound holds when the 3 points $\vec{r}$, $\vec{p}_i$, and $\vec{p}_j$ are collinear, and $\vec{p}_i$ and $\vec{p}_j$ are on the same side of $\vec{r}$. The equality limit of the upper bound holds when the 3 points are collinear and $\vec{p}_i$ and $\vec{p}_j$ are on opposite sides of $\vec{r}$.

Proposition 1

$$|R_k - R_i| > D_{kj} \Longrightarrow D_{ki} > D_{kj} \tag{3}$$

Proof : Reversing the lower bound of the triangle inequality gives $D_{ki} \geq |R_k - R_i|$. Substituting this into the left side of expression (3) gives $D_{ki} \geq |R_k - R_i| > D_{kj}$, which yields $D_{ki} > D_{kj}$.



a) E_c vs. N

b) E_t vs. N

Figure 4: Performance for $1 \leq n \leq 500$

4.1 TINN Method

A novel solution to the Nearest Neighbour Problem based upon Proposition 1, which we call the *Triangle Inequality Nearest Neighbour (TINN)* method, is as follows.

1. The point set is first preprocessed by calculating R_i for all $\vec{p}_i \in \mathbf{P}$. The results are organized into a list $\mathbf{L}$ such that each list element L_i references a point and its distance to the reference point, i.e. $L_i = \{\vec{p}_i, R_i\}$. The list is then ordered by increasing R_i. We will assume in the subsequent description that the subscript enumerates the ordering of $\mathbf{L}$, i.e. $i < j \iff R_i < R_j$.
2. At run time, the value R_q is determined for the query point $\vec{q}$. The list element whose distance to the reference point is closest to R_q is then identified and denoted as the *pivot* element L_k. This can be accomplished efficiently as a binary search over $\mathbf{L}$. The distance D_k between the query point and the pivot point is calculated, and the closest point is temporarily assigned as the pivot point, i.e. $\vec{p}_c \leftarrow \vec{p}_k$, $D_c \leftarrow D_k$.
3. Starting at the next lowest neighbour $\vec{p}_{k-1}$ of $\vec{p}_k$, we then compare the value of $|R_q - R_{k-1}|$ with D_k. If $|R_q - R_{k-1}| \leq D_c$, we then calculate D_{k-1}. If $D_{k-1} < D_c$, then we reset the temporary closest point to $\vec{p}_{k-1}$, i.e. $\vec{p}_c \leftarrow \vec{p}_{k-1}$, $D_c \leftarrow D_{k-1}$. Otherwise, if $|R_q - R_{k-1}| > D_c$, then there is no need to calculate D_{k-1} as Proposition 1 guarantees that its value will exceed D_c and that $\vec{p}_{k-1}$ cannot be the closest point. Furthermore, it is unnecessary to consider any other list elements in the same direction, as the ordering of $\mathbf{L}$ ensures that $|R_i - R_q|$ will increase for all monotonic sequences of i on the same side of L_k.
4. Step 3 is repeated for all neighbours in the same direction (i.e. $\vec{p}_{k-h}, h > 1$), until either the stopping condition is met ($|R_q - R_{k-h}| > D_c$), or the top of $\mathbf{L}$ is reached.
5. Steps 3 to 4 are repeated for the L_i on the other side of the original pivot element (i.e. $\vec{p}_{k+h}, h \geq 1$).

The time expense to generate $\mathbf{L}$ is small, requiring only a single pass through $\mathbf{P}$ to determine the R_i, followed by a sort. The storage cost is also extremely modest, comprising an extra float value per point, plus a single reference point $\vec{r}$. The method also trivially extends to solve the K Nearest Neighbour Problem, and to higher dimensional metric spaces.

A geometric interpretation of the method is illustrated in Figure 5 with a simple 2D example. Part a) shows a 4 element point set, reference point $\vec{r}$, and query point $\vec{q}$. In part b), the distances R_i to $\vec{r}$ of $\vec{q}$ and all $\vec{p}_i$ are shown. In part c), $\vec{p}_1$ is determined to be the current pivot point. Only those points which lie within the annulus of radius $R_q \pm D_1$ centered at $\vec{r}$ need be considered any further. The next point in $\mathbf{L}$ is $\vec{p}_2$, which is determined to fall within the annulus. As $D_2 < D_1$, $\vec{p}_2$ is set to the current pivot point. In part d), it is shown that all remaining points fall outside of the new annulus, and the process terminates having identified $\vec{p}_2$ as the nearest neighbor. It was necessary to explicitly calculate D_i for only 2 of the 4 points, yielding an efficiency of $E_c = 2$.

4.2 Performance Comparison

Versions of the TINN, Elias, and k-d tree methods were implemented and tested on a number of uniformly distributed random point sets. Each implementation was executed on the same platform and compiled with the same compiler options, and a moderate and equivalent amount of attention was afforded to the optimization of each. In each test, the

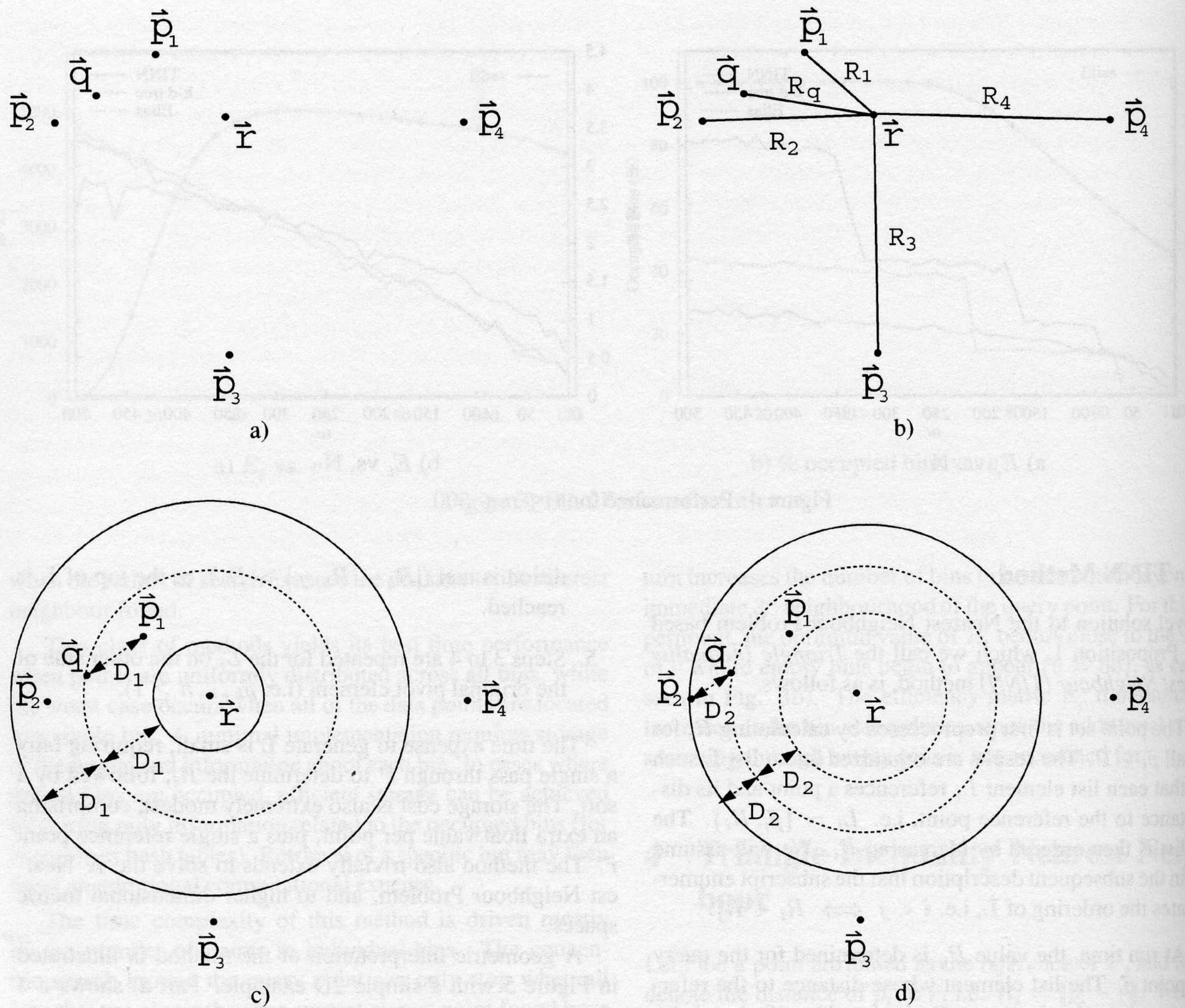


Figure 5: Geometric Interpretation

lowest corner of the bounding box of **P** was selected as the reference point $\vec{r}$ for the TINN method.

It was clear from some cursory tests that, compared to k-d tree and Elias methods, TINN was not particularly effective for large point sets. For small point sets, however, TINN outperformed both. Figure 4 plots the results of E_c and E_t for a test of 10^4 uniformly randomly distributed query points with point sets of size $1 \leq n \leq 500$. It can be seen that TINN exhibits superior time performance for $n < 275$, even though its E_c value only exceeds those of the k-d tree and Elias methods for $n < 50$. Indeed, TINN outperforms a fairly lean implementation of the exhaustive search method for just $n > 40$ (i.e. $E_t > 1$ for $n > 40$). For $n >> 275$ all curves continue upwards, and both k-d tree and Elias continue to outperform TINN for both metrics.

5 Hybrid Methods

The above performance results lead to a hybridization scheme which is applicable to both k-d tree and Elias, or indeed any binning method. The approach is to use relatively coarse bins, so that the number of points resident in a bin is greater than 40. When a bin is interrogated, the TINN method is applied in the place of exhaustive searching.

5.1 K-d Tree / TINN

In Figure 6, the time performance of the hybrid k-d tree/TINN method is compared with that of the pure k-d tree for a point set of $n = 10^6$. From this result, three benefits of the hybrid method can be identified:

1. **Time Efficiency:** The hybrid method is more time efficient than the k-d tree. At their respective optimal values of B, the E_t values indicate that the hybrid method performs $\sim 20\%$ faster than the k-d tree. The E_c value of the hybrid method is greater than that of the k-d tree for all values of B.

2. **Space Efficiency:** The optimal E_t for the k-d tree occurs at bin size $B \approx 20$. For the hybrid method, this same value of E_t is realized for up to $B = 400$. As the total number of nodes of a binary tree is twice the number of leaf nodes, the hybrid method therefore achieves the same time performance for a tree which is 20 times smaller.

3. **Sensitivity:** The slope of the E_t vs. B curve is more gradual for the hybrid method. This increases the likelihood of selecting a value of B for which E_t is close to optimal for an unknown point set.

5.2 Elias / TINN

In Section 3 we described how, as the bin size increases (i.e. v decreases), the cost of the exhaustive search in each examined bin becomes the dominant limit to performance. Larger bins are therefore desirable not only for reducing memory consumption, but also for minimizing the proportion of empty bins. In the Elias method, the exhaustive search phase within a bin can be replaced by any technique for finding the nearest neighbour among a set of points. Given a sufficiently large point set in a subregion, the TINN method will provide an increase in performance over exhaustive search.

The implementation used in the experiments of Section 3 was modified to apply a TINN search over each bin, the rest of the code remaining identical. Figure 7 shows the relative performance of the hybrid and basic Elias method. As expected, the hybrid method outperforms Elias for large bins. For example, at $v = 15$, where the average number of points in a bin is 125, the E_t value for Elias is 1455 compared to 3205 for the hybrid. The overall peak performance of the hybrid, however, is slightly less than the peak of the basic Elias method.

As seen in Fig. 7a), the hybrid method always outperforms the basic one in E_c, since the TINN will always be at least as efficient as the exhaustive search within each bin. The performance improvement is highest near $v \approx 40$.

6 Conclusion

We have described the TINN method, which is a new solution to the Nearest Neighbour Problem. On its own, TINN only works well for very small point sets. When TINN is applied to the bin sets of either k-d tree or Elias methods, their performance is enhanced. For large point sets, the k-d tree hybrid exhibits both improved time performance and a reduced memory expense. While the Elias hybrid did not improve the absolute time performance, the memory consumption was drastically reduced at only a slight time performance penalty, a feature which is attractive for dealing with large point sets.

The experiments described here were done on uniform random distributions of points in the volume. If the points were distributed otherwise, for example on an embedded surface, then only very large bins would limit the proportion of empty bins, which would cause rapid degradation of the time performance of the basic Elias method, thereby further motivating the use of the hybrid method. A similar argument holds for the k-d tree, where a uniform random distribution results in a very balanced tree structure. Other distributions can result in an unbalanced structure, which would result in increased backtracking.

In this work, we have based our arguments on a direct evaluation of the run time performance of implementations of the various methods. Such an approach can be problematic, as it is dependant upon implementation specifics, such as coding and machine architecture. We believe that, while the operating points may vary for different implementations, the principles described within this paper will persist. For example, an extremely efficient implementation of the exhaustive search may increase the minimum value of n for which TINN is effective. TINN may also be substituted for any suitable alternative methods which perform well for small point sets.

Finally, the hybridization approach seems likely to enhance performance to an even greater degree for increasingly large point sets.

Acknowledgements

We would like to thank the reviewers for their insights and helpful comments.

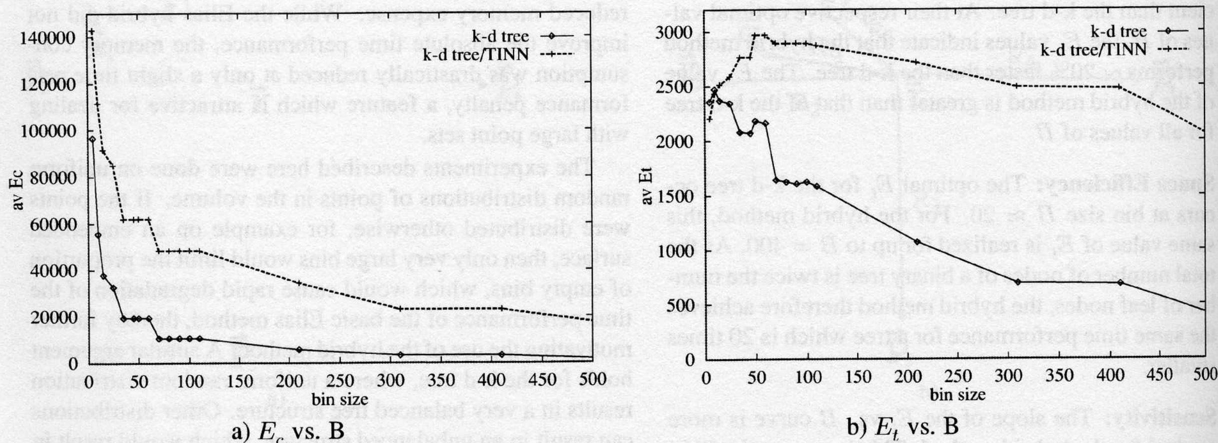
References

[1] Paul J. Besl and Neil D. McKay. A method for registration of 3d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[2] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

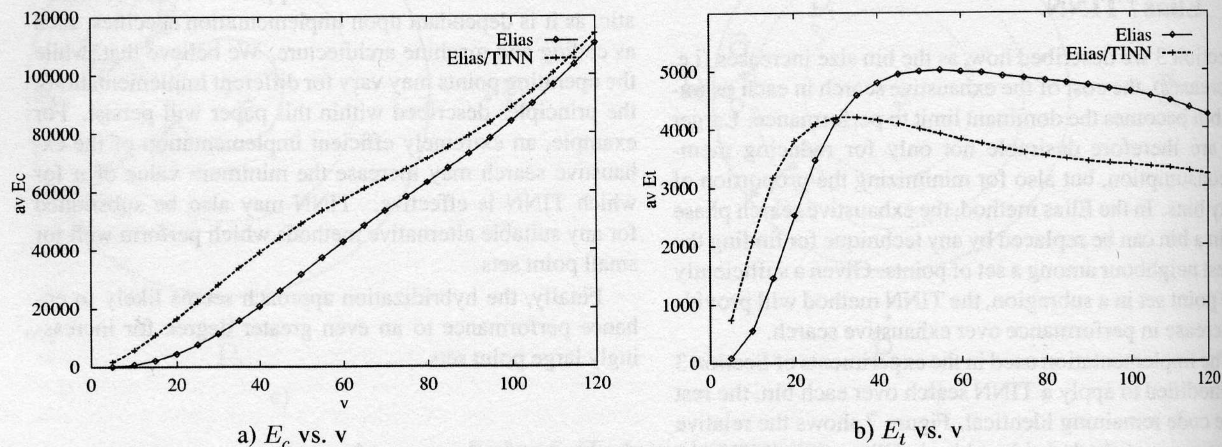
[4] Ronald L. Rivest. On the optimality of elias's algorithm for performing best-match searches. In *Information Processing 74*, pages 678–681, 1479.



a) E_c vs. B

b) E_t vs. B

Figure 6: Comparison of K-d Tree and K-d Tree/TINN



a) E_c vs. v

b) E_t vs. v

Figure 7: Comparison of Elias and Elias/TINN

[5] John Gerald Cleary. Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Transactions on Mathematical Software*, 5(2):183–192, June 1979.

[6] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[7] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, pages 750–753, July 1975.

[8] W.A. Burkhard and R.M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[9] Enrique Vidal Ruiz. An algorithm for finding nearest neighbors in (approximately) constant time. *Pattern Recognition Letters*, 4:145–157, July 1986.

[10] Enrique Vidal, Hector M. Rulot, Francisco Casacuberta, and Jose-Miguel Benedi. On the use of a metric-space search algorithm (aesa) for fast dtw-based recognition of isolated words. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(3):651–660, May 1988.

[11] Sameer A. Nene and Shree K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, September 1997.

[12] Robert F. Sproul. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.

[13] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.