Simple and Fast Nearest Neighbor Search

Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler*
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
marcelbirn@gmx.de, holtgrewe@ira.uka.de, {sanders,singler}@kit.edu

Abstract
We present a simple randomized data structure for two-dimensional point sets that allows fast nearest neighbor queries in many cases. An implementation outperforms several previous implementations for commonly used benchmarks.

1 Introduction

Given a set $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n points in two-dimensional Euclidean space, we want to build a linear space data structure that can answer nearest neighbor queries: Given a point $q = (x, y)$, return the point $p \in S$ that minimizes the Euclidean distance $||p - q||_2$. This operation is important, e. g., in geographical databases. In a route planning system, when a user clicks on a map, she may want to find the nearest road junction in the network. Solutions with logarithmic query time are known but they are complicated and have large constant factors in their running time. Therefore, practical solutions often use algorithms without worst-case-optimal behavior. In Section 2, we give yet another such technique – Full Delaunay Hierarchies (FDH) – which is even simpler and very fast in practice. We also obtain analytical evidence for logarithmic query time. The experiments described in Section 3 indicate that our algorithm can compete with all previous implementations, and outperforms them by a considerable factor in some situations.

Related Work. The most popular way for doing nearest neighbor queries is using variants of kd-trees [3, 11]. Although they do not give good worst case guarantees, they work well for many practical input distributions, and for approximate queries. Linear space and logarithmic query time can be achieved via point location in the Voronoi diagram of the point set [6]. However, this approach does not seem to be used in practice. In particular, it might be expensive to implement this with exact arithmetics since the vertices of the Voronoi diagram are only implicitly defined by the center of the circumcircle of three input points.

There are several results similar to ours. The hierarchy of Delaunay triangles used in [9] (which is also similar to [4]) is kind of dual to our FDH, which consists of nodes. We expect our approach to be faster on the average since testing whether a point is inside a triangle is more expensive than comparing Euclidean distances. The Delaunay hierarchy (DH) used in [7] uses a nested collection of logarithmically many geometrically growing levels $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_k = S$, running a point location query in each level. FDHs can be viewed as DHs with n levels. FDHs are also simpler than DHs because we only compare point distances, whereas DHs involve three subphases for every level, involving both point locations and walks through triangles along rays towards the query point. What makes FDHs interesting is that they at the same time simplify the algorithm and accelerate queries in practice. Our motivation for trying this was the successful transition from *highway-node routing* [12] to *contraction hierarchies* [8] in route planning, where going from $\mathcal{O}(\log n)$ levels to n levels brought both a significant simplification and an order of magnitude improvement in query time.

A disadvantage of techniques based on Delaunay triangulations is that they are unlikely to work well in higher dimensions. Unfortunately, seemingly related graphs that remain sparse for higher dimensions do not have the properties we need. For example, the well known Yao-graph [13] allows a greedy algorithm like the one considered in the next section, and even allows logarithmic path length [2] – but this only works if the query point actually is in the input set, otherwise it might give wrong results.

2 Full Delaunay Hierarchies

Assume the points $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ have been randomly numbered $1..n$.[1] Let the graph $G_k = (1..k, E_k)$ denote the Delaunay triangulation of points $1..k$. Nearest neighbor search with respect to points $1..k$

*Partially supported by DFG grant SA 933/3-2.

[1] We use $a..b$ as a shorthand for $\{a, \ldots, b\}$.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

can be done using a well-known greedy algorithm [10]: Starting from any node u, consider its neighbors v in the Delaunay triangulation. If v is closer (to the query point q) than u, then set $u := v$. Iterate until no neighbor gets closer. A *full Delaunay hierarchy* (FDH) of S is the graph $G = (1..n, \bigcup_{k=1}^{n} E_k)$. The required information can be recorded while building the graph for $1..n$ using randomized incremental construction. This takes expected time $\mathcal{O}(n \log n)$ and the expected number of edges in G is $\mathcal{O}(n)$ [9]. We now show that a simple greedy algorithm efficiently finds the nearest neighbor. The algorithm works on the FDH looking only at *downward* edges, i. e., edges towards nodes with larger index. Figure 1 gives pseudocode, Figure 2 shows an example run.

THEOREM 2.1. *The algorithm from Figure 1 outputs a nearest neighbor. The expected number of nodes traversed is bounded by* $\ln n + 1$.

Proof. We show by induction that the algorithm implicitly traverses a sequence σ of points that are the nearest neighbors of q with respect to 1..1, 1..2, ..., $1..n$. For the basis of the induction, observe that by initializing u with 1 we fulfill the claim for 1..1. Now, suppose the algorithm has found a prefix $\langle 1, \ldots, u \rangle$ of σ such that u is closest (to q) with respect to $1..k$. To extend the induction to $k+1$, we have to consider two cases:
Case: u is also closest with respect to $1..k+1$.
Even if there is an edge $(u, k+1)$, the algorithm will stick to u.
Case: $k+1$ is closer than u.
From the greedy algorithm, we know that there must be a path from u to $k+1$ in G_{k+1} with nodes that are monotonically getting closer. By definition of an FDH, this path is also part of the FDH. Moreover, by the induction hypothesis, this path can consist of at most one edge, i. e., it has the form $e = (u, k+1)$. Since $u < k+1$, e is a downward edge, i. e., e is considered by Algorithm 1, and so $k+1$ is the next element in the sequence of points considered by the algorithm.

For the path length, we need to exploit the random order of the nodes. Let X_i denote the indicator random variable that is 1 iff node i appears in σ. This happens iff i is nearest with respect to $1..i$. The probability for this event is $1/i$. The total path length is $|\sigma| = \sum_{i=1}^{n} X_i$, i. e.,

$$\mathbf{E}[|\sigma|] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = H_n \leq \ln n + 1 \ .$$

Note that we exploited the linearity of expectation and a well-known bound for the harmonic sum H_n. ■

```
Procedure nn(q)
    u := 1                          -- start at top of hierarchy
    repeat
        foreach neighbor v of u in G
                with u < v in increasing order do
            if ||q - s_v||_2 < ||q - s_u||_2 then
                u := v
                break foreach-loop
    until no improvement found
    return u
```

Figure 1: Nearest Neighbor Query in an FDH.

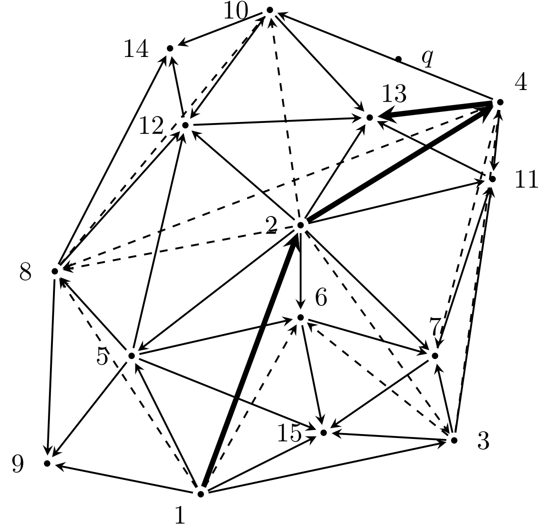

Figure 2: Finding the nearest neighbor of q using FDH. Solid edges belong to the final Delaunay triangulation, while the dashed ones stem from the incremental construction.

The total path length is also a good indicator for the number of cache faults[2] because our implementation will implement the inner loop so that only a contiguous memory area needs to be scanned. Unfortunately, we can not easily bound the execution time in a similar way since already the Delaunay triangulation may contain nodes of very high degree, about half of which will have to be traversed in the FDH.

We get bad performance if there are nodes with high out-degree in the FDH. For example, consider the input $\{(0,0)\} \cup \{\cos(2\pi i/n), \sin(2\pi i/n) : i \in 1..n\}$ – the origin plus n points equally spaced on the unit circle. The

[2]On some machines, this will even work when nodes on the path have very large degrees since the resulting memory accesses allow data prefetching.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

center point $(0, 0)$ has degree n in the Delaunay triangulation and its expected out-degree after randomized incremental construction is $n/2$. Moreover, a random query point in the circle has constant probability to incur a path through the FDH that leads over the center point, i. e., we have average case query time $\Theta(n)$ for this instance.

2.1 Finger Search. In some applications, search can be accelerated if we already know a *finger* f that is supposedly close to the query point q. A *nearest neighbor finger query* is split into an upward phase and a downward phase. Rather than starting from the topmost node 1, we start from f. In the upward phase, we look for edges of the FDH that lead to nodes with smaller index closer to q. When this is no longer possible at some node u, we know that u is closest to q with respect to $1..u$ (Claim 1). Since this property is the invariant needed for Algorithm 1, we can now switch to a downward phase that will find the nearest neighbor.

Proof of Claim 1: Suppose there is a set N of nodes in $1..u$ that are closer to q than u. Consider the circle C with center q such that u lies on the boundary of C. By the definition of N, all nodes in N are inside circle C. Now we conceptually move the center of C towards u, shrinking the circle so that u remains on the boundary of C. One by one, the points in N will drop out of C. Consider the point t that drops out last. When this happens, both u and t lie on the boundary of C and no point in $1..u$ lies inside C. Hence, by a well-known characterization of Delaunay triangulations (e. g. [6, p. 189]), the edge (u, t) is in the Delaunay triangulation of $1..u$. This contradicts our original assumption and thus there cannot be any nodes in $1..u$ that are closer to q. ∎

Finger search can also be used to speed up non-finger queries: We can use a simple and fast heuristics to find a point f that is "usually" close the nearest neighbor. We can then use f as a finger information to obtain the actual nearest neighbor using a more robust algorithm. In particular, we can use nearest neighbor search with inexact arithmetics in combination with exact finger search to speed up exact finger search – in most cases, the inexact search will have uncovered the true nearest neighbor and the exact finger search only has to verify that looking at the outgoing edges of a single node of the FDH.

2.2 Implementation. The query algorithm is dominated by scanning downward edges. Since the distance to q needs to be compared with the current distance, it is advantageous to store the coordinates of the endpoints of an edge together with the edge. In any case, the outgoing edges of each node are stored with target nodes ascending, so the query algorithm can just scan the edges in this order.

Note that this optimization implies a graceful degradation for bad inputs: Even for worst case inputs where we may have to scan $n - 1$ outgoing edges, the computations performed are basically the same as in the naive algorithm where we simply have to scan an array of n points to find the closest. In contrast, other algorithms perform complicated operations in pointer based data structures, which incur many cache faults and can therefore become much *slower* than the naive algorithm.

We have implemented FDHs using the CGAL library [5] (version 3.4).

For the construction, we slightly deviate from the basic blueprint of randomized incremental construction. We adopt the idea of spatial sorting from [1] to split the randomly permuted input sequence into levels shrinking by a factor of four in each step, and sorting each level spatially using a Hilbert curve. This functionality is provided by `CGAL::spatial_sort`. We then insert the points into a Delaunay triangulation (not a hierarchy).

The program does not compare Euclidean distances but their squares since that way, no square roots need to be evaluated. In the exact implementation of FDHs we first perform an inexact distance computation using interval arithmetics. If the outcome might be too imprecise, we use an exact predicate of CGAL. Furthermore, we exploit that one of the distance intervals is known from the previous iteration. This solutions proved faster than CGAL's built-in `Compare_distance_2`.

3 Experiments

For FDHs, there are variants for exact arithmetics and plain floating point arithmetics (label "inexact" in the figures). Furthermore, there are variants with and without the optimization from Section 2.2, namely storing coordinates of target points together with the edge array (label "array").

We compare ourselves with two implementations using kd-trees that use inexact arithmetics: one available within CGAL, and the other from [11], which was recommended to us as the currently fastest nearest neighbor code. Finally, we compare to the CGAL implementation of Delaunay hierarchies with exact arithmetics.

We use the same inputs as in [7]:

1. Random points in the unit square with random query points from a square that is 5 % larger than the square with input points.
2. Random point on the border of the unit circle with random query points from the smallest square containing the circle.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

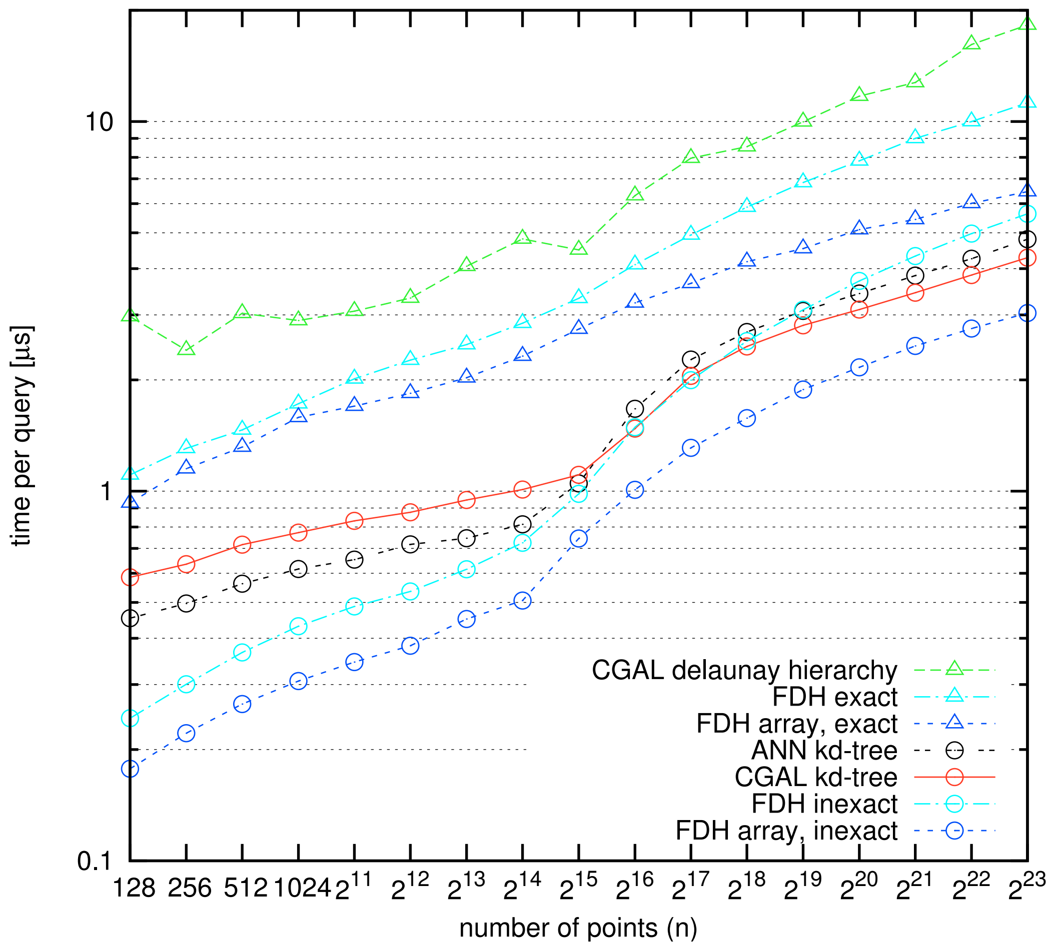

Figure 3: Query performance for random points in the unit square. See Table 1 for numerical values.

3. Random x coordinate from $[-1000, 1000]$ and $y = x^2$ (parabola) with random query points from a rectangle containing that part of the parabola.

4. A mix of 95 % of the points on the border of the unit circle and 5 % random points from the smallest square enclosing the circle. The query points are random points from the unit circle.

Our implementations are compiled using the C++ compiler g++ 4.3.2 with options `-O3 -frounding-math` and run on one core of a 2.33 GHz Intel Xeon E5345 with a 4 MByte cache and 16 GByte of main memory.

Figure 3 shows the average query time for all the implemented algorithms. The FDH algorithm with the coordinates in the edge array (label "array") uniformly outperforms the FDH algorithm without this enhancement. The gain is considerable, except for the exact variant and small inputs where it is plausible that cache faults are not yet an issue, and the overhead for exact arithmetics dominates execution time. Although this runtime improvement comes with some space overhead, from now on we focus on the fast variant with coordinates in the edge array. This algorithm outperforms the kd-tree algorithms by more than a factor of two for small inputs and by at least 30 % for large inputs. The relative performance of the kd-tree codes depends on the input size. The ANN implementation is better for small inputs whereas the CGAL implementation is slightly faster than ANN for large inputs. Again, cache effects may be an explanation. Among the exact algorithms, FDH is also uniformly faster than its competitor Delaunay hierarchies. The lead is more than a factor of two for small inputs and always at least 30 %. There is a factor of 2–3 performance difference between the exact and inexact implementations of FDHs. Note that such a performance difference is to be expected even if few exact computations are necessary since interval arithmetics is considerable more expensive then

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

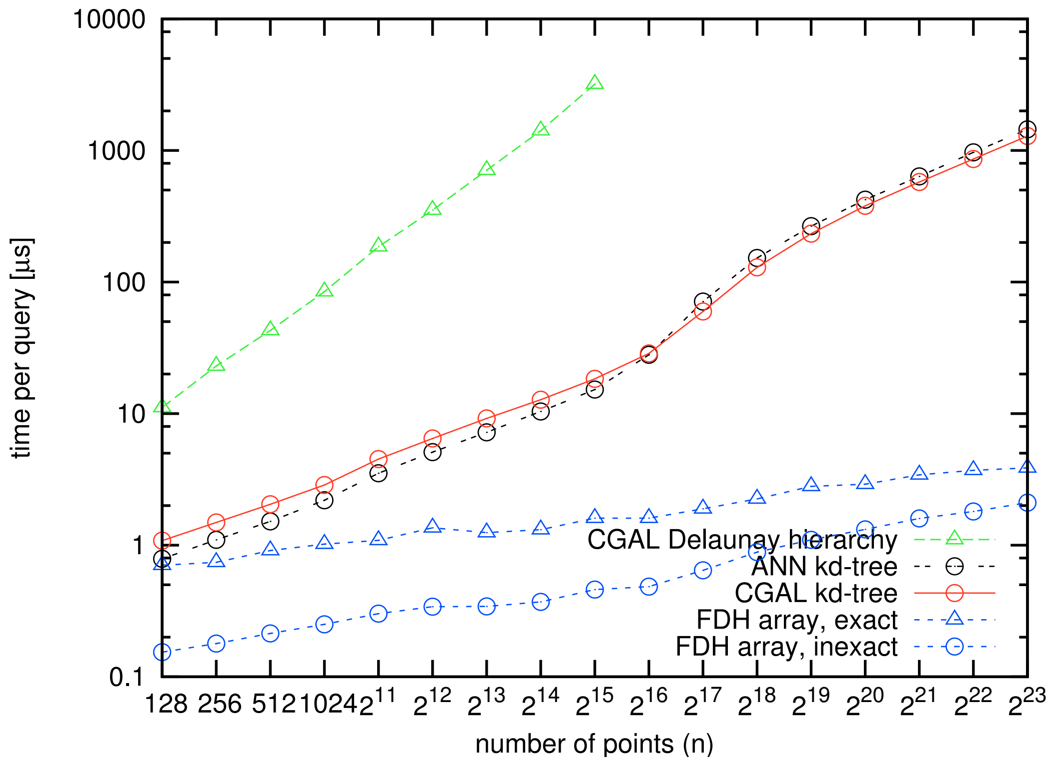

Figure 4: Query performance for random points on a circle. See Table 2 for numerical values.

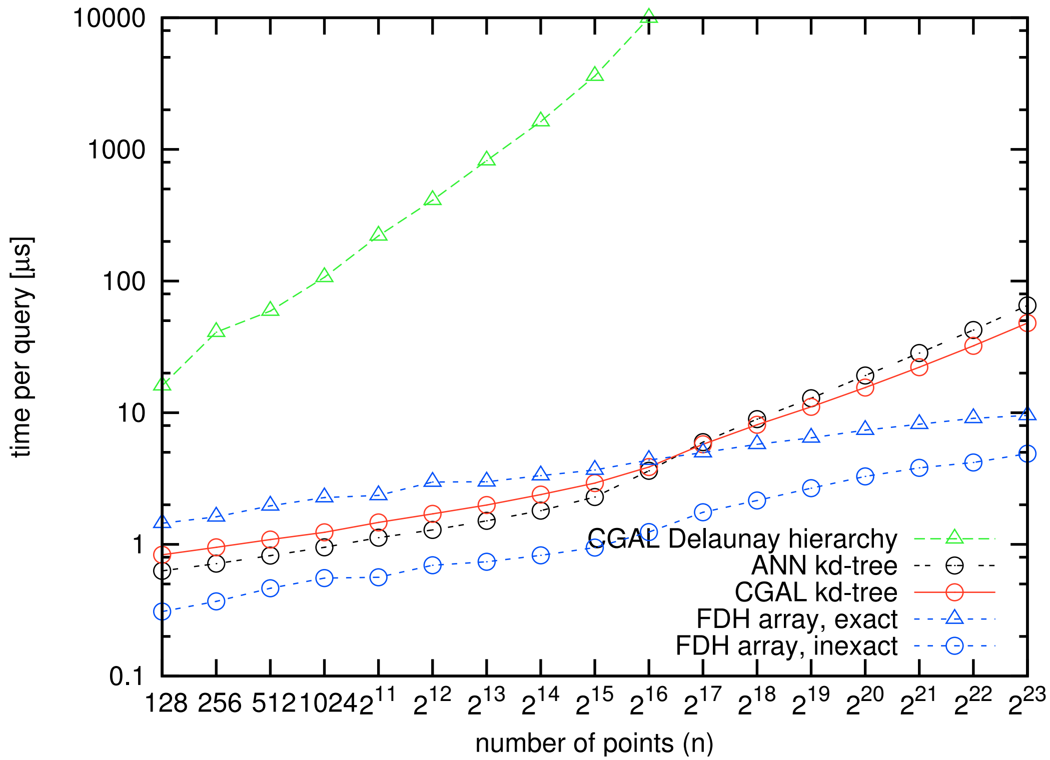

Figure 5: Query performance for random points on a parabola. See Table 3 for numerical values.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

ordinary arithmetics, and since our code does little else than distance calculations.

This looks like a significant yet moderate lead for FDH compared to its competitors. However, for other inputs the situation looks completely different. Figure 4 shows the query performance for input points on a circle. Both the kd-trees and the Delaunay hierarchy do not scale well with the input size. For the largest inputs, FDHs are several hundred times faster than their competitors. Additional experiments not shown here explain the bad performance of the Delaunay hierarchy – finding the right facet is very fast, but finding actual closest points takes almost all the time. For the parabola instances (Figure 5), FDHs are somewhat slower than before but still win (comparing exact algorithms only with exact competitors). For large instances, this lead is by a large margin. Indeed, Delaunay hierarchies are catastrophically slow.

For mixed inputs, FDHs still outperform Delaunay hierarchies but they are beaten by kd-trees (Figure 6). The problem is that the points are lying denser on the circle than in the rest of the square. Thus, points close to the circle will have very high degree in the Delaunay triangulation. Even worse, the same will happen on all levels of the FDH so that we can get high degree during the entire search.

Figure 7 evaluates the performance of finger queries for 2^{23} random points in the unit square. We use random query points and a finger point that is chosen randomly on a circle around the query point. By varying the radius of this circle, we can vary the quality of the finger information, which we express by the expected number of input points that are closer to the finger than the query point. We compare two variants of using finger information: trying the steepest edges first, i. e., going to the node highest up in the hierarchy first, and the opposite strategy of trying the most shallow edges first. As to be expected, when the finger is close to the point we are looking for finger queries perform better than queries that ignore finger information. When the finger information is not good, we get an overhead up to a factor of ≈ 2, because in the worst case, we have to go all the way up the FDH and then down again. The results on the relative performance of steep versus flat variants is ambivalent. On the one hand, the flat strategy performs somewhat better when the finger is close to the result point. Since many applications will use finger search only in this situation, we might conclude that the flat strategy should be used. On the other hand, the steep strategy is considerably better if the finger information has low quality. Hence, we should perhaps prefer the steep strategy as a default since it is more robust. The break-even point between the algorithms with and without finger information differs between exact and inexact arithmetics. Theoretically, we would have expected a break even point around $\sqrt{n} \approx 3000$ points closer to the query point. For inexact arithmetics, the break even point is lower and for exact arithmetics it is somewhat higher, in particular for the steep strategy.

By using inexact queries to initialize inexact finger search we can accelerate exact search as can be seen in Figure 8.

We have not invested a lot of effort in fast construction or space efficiency, i. e., we use the CGAL code for randomized incremental construction of Delaunay triangulations, and store the query data structures for FDH and kd-trees separately. In particular, it should be noted that our speedup techniques of storing target coordinates in the edge array and using inexact FDHs and finger search to speed up exact search, incur a price in additional preprocessing time and space. Thus, the results reported in Figure 9 should be taken with a grain of salt. We can see that constructing the FDH data structure puts a moderate additional overhead on top of a considerable cost per point for constructing the Delaunay triangulation.

All the depicted algorithms have asymptotic running time $\Theta(n \log n)$ which should translate into a straight line in the semi-logarithmic plot. Interestingly, the practical running time seems to be dominated by a linear term in most cases. Only the kd-trees have a quite steep increase of construction time once the data structure does not fit into the cache. We do not show construction times for the other inputs since the construction time does not depend on the inputs as much as the query time.

We do not show space overhead for the different variants since, on the one hand, space overhead is easy to derive for a particular implementation, and, on the other hand, space consumption depends on a lot of details like layout rules of the compiler, the data type used for coordinates, and how one counts the space used during construction. Hence, the relative space consumption of different approaches could go in either direction depending on several implementation details.

4 Conclusions

Full Delaunay Hierarchies are a simple and fast data structure for nearest neighbor queries in two dimensional point sets. For many input distributions, they are the fastest currently available implementation both for exact and inexact arithmetics. It is generally interesting that we gave an additional example where hierarchical data structures with a level of hierarchy for each input object by far outperform their competitors with a small

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

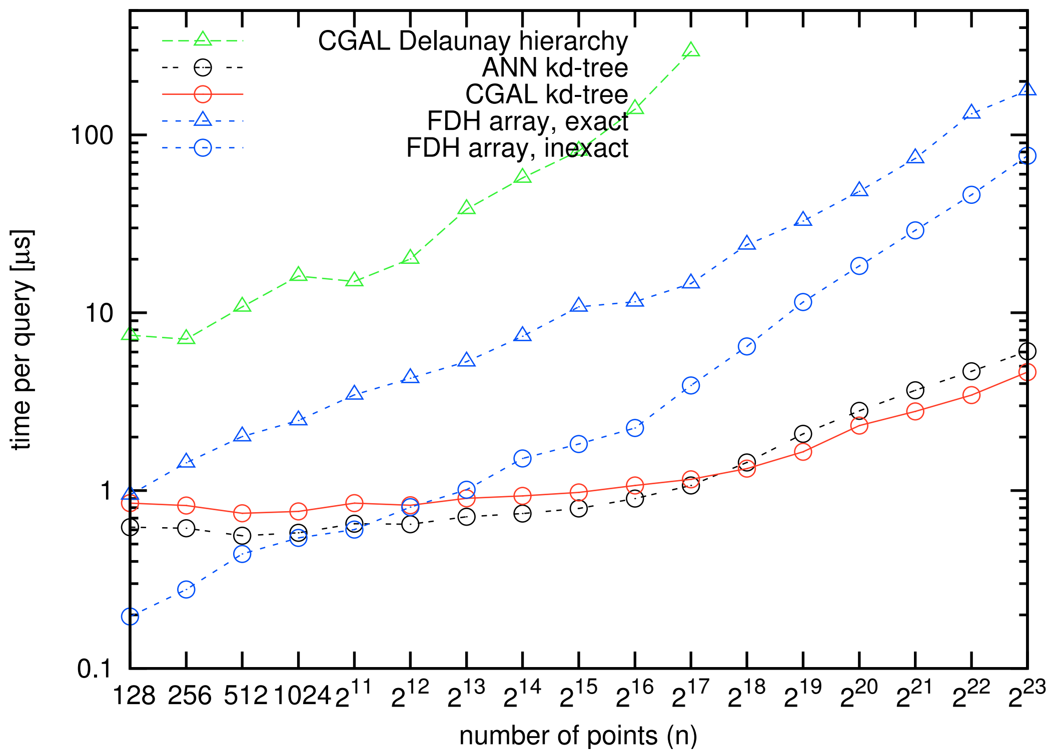

Figure 6: Query performance for mixed inputs circle / square. See Table 4 for numerical values.

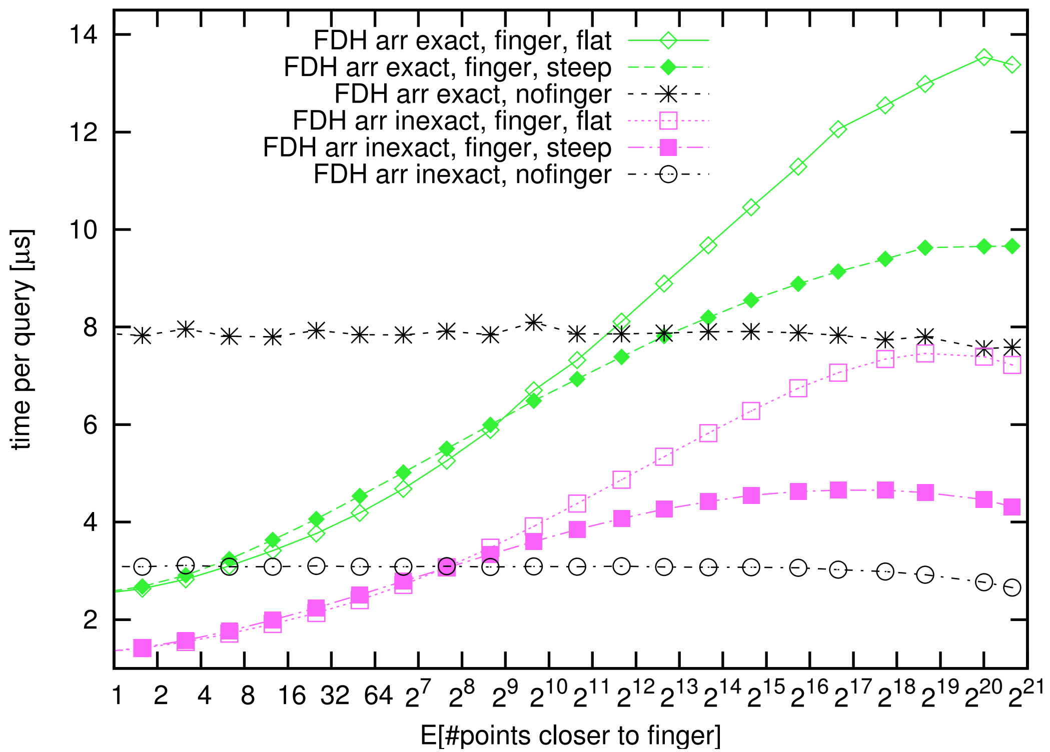

Figure 7: Query performance of finger queries on 2^{23} random points in a square. See Table 5 for numerical values.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

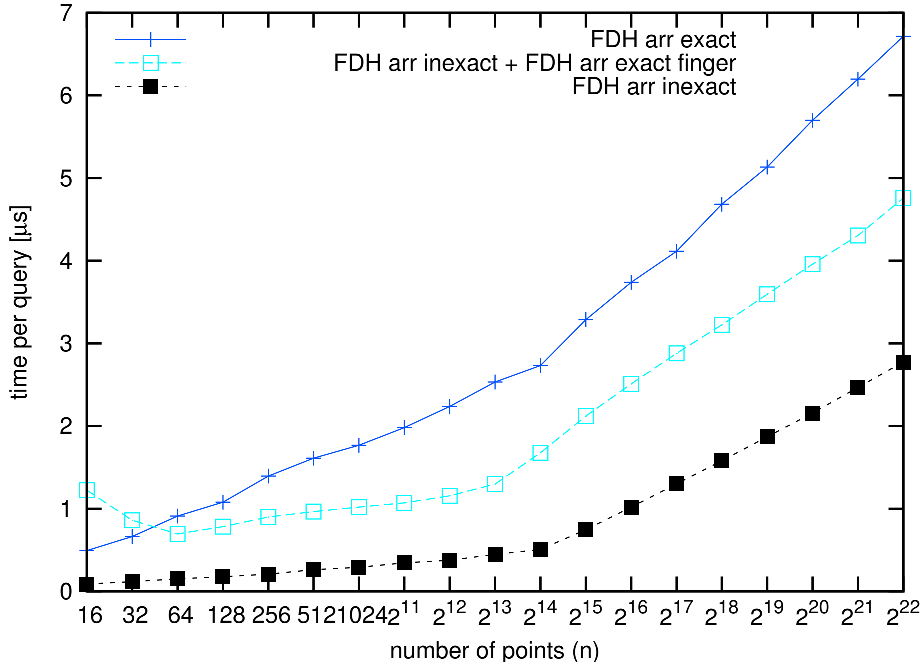

Figure 8: Using finger search to speedup exact non-finger search for random points in a square. See Table 6 for numerical values.

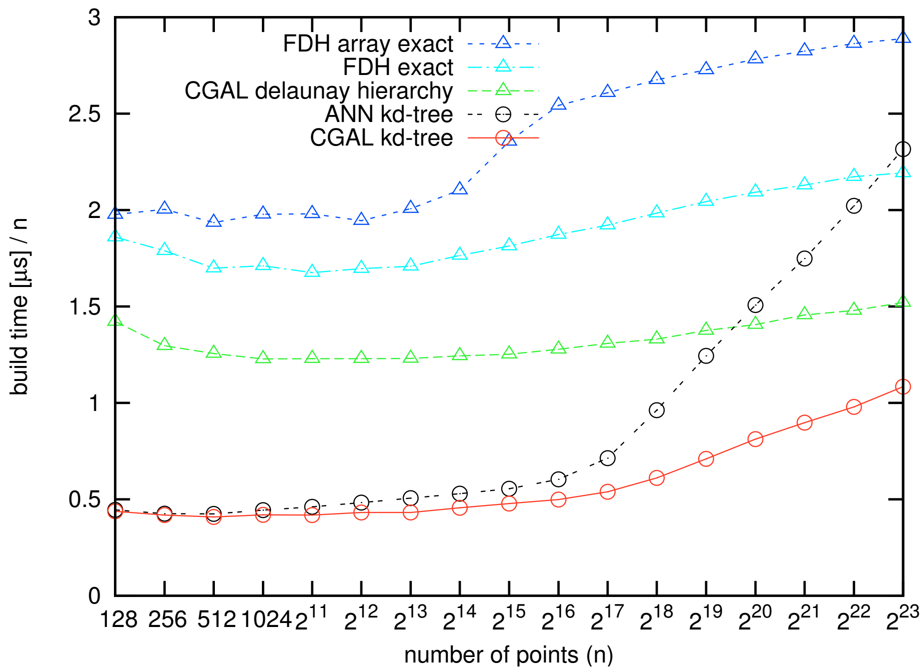

Figure 9: Construction time per random point in a square. See Table 7 for numerical values.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

number of levels (Delaunay hierarchies [7] in this case).

A major open question for the two-dimensional nearest neighbor problem is a practical algorithm with worst case logarithmic query time. We considered several approaches based on FDHs but without finding a provably good solution so far. One interesting idea is to go away from a random ordering of the nodes. For example, the worst case input from Section 2 becomes an easy input if the point of degree n in the Delaunay triangulation is inserted *last* – it then has high *indegree* in the FDH, and this is no problem for the query algorithm. Another interesting idea is to replace high degree nodes in the FDH by several close-by points using the concept of symbolic perturbation. A nice property of such a solution would be that additional overhead is only incurred locally in regions of the input where the points are unevenly distributed. In contrast, using Voronoi diagrams incurs a constant factor overhead everywhere (For example, every vertex of a Voronoi diagram is an object implicitly defined by three input points and many point location data structures would additionally triangulate the Voronoi diagram.).

Acknowledgments. We thank Vitaly Osipov and Ludmila Scharf for helpful discussions.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *19th Symposium on Computational Geometry*, pages 211–219, 2003.

[2] S. Arya, D. M. Mount, and M. H. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *35th Symposium on Foundations of Computer Science*, pages 703–712, 1994.

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

[4] J.-D. Boissonnat and M. Teillaud. The hierarchical representation of objects: The Delaunay tree. In *ACM Symposium on Computational Geometry*, pages 260–268, 1986.

[5] CGAL consortium. Computational geometry algorithms library. `http://www.cgal.org/`.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[7] O. Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13:163–180, 2002.

[8] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008.

[9] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[10] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *11th Canadian Conference on Computational Geometry*, 1999.

[11] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. *CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997.

[12] D. Schultes and P. Sanders. Dynamic highway-node routing. In *6th Workshop on Experimental Algorithms (WEA)*, pages 66–79, 2007.

[13] A. C.-C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Input Size	CGAL D. hierarchy	FDH exact	FDH array exact	ANN kd-tree	CGAL kd-tree	FDH inexact	FDH array inexact
128	2.968	1.106	0.929	0.453	0.585	0.243	0.177
512	3.027	1.465	1.317	0.562	0.716	0.366	0.266
2048	3.067	2.012	1.698	0.652	0.831	0.488	0.345
8192	4.059	2.492	2.026	0.746	0.946	0.615	0.451
32768	4.496	3.322	2.743	1.050	1.108	0.984	0.745
131072	7.958	4.932	3.649	2.272	2.050	1.995	1.310
524288	9.990	6.839	4.535	3.076	2.808	3.093	1.885
2097152	12.752	9.008	5.435	3.829	3.441	4.328	2.472
8388608	18.249	11.245	6.464	4.802	4.280	5.630	3.034

Table 1: Time per query [µs] for random points in the unit square.

Input Size	CGAL D. hierarchy	ANN kd-tree	CGAL kd-tree	FDH array exact	FDH array inexact
128	11.130	0.787	1.083	0.705	0.154
512	42.783	1.515	2.042	0.910	0.213
2048	185.387	3.526	4.505	1.088	0.302
8192	707.881	7.213	9.184	1.248	0.342
32768	3191.392	15.249	18.395	1.602	0.460
131072	–	70.927	59.696	1.886	0.644
524288	–	266.045	233.425	2.794	1.097
2097152	–	634.556	575.258	3.431	1.600
8388608	–	1449.437	1285.948	3.873	2.101

Table 2: Time per query [µs] for random points on a circle.

Input Size	CGAL D. hierarchy	ANN kd-tree	CGAL kd-tree	FDH array exact	FDH array inexact
128	16.043	0.630	0.832	1.449	0.309
512	59.280	0.822	1.088	1.961	0.464
2048	220.582	1.126	1.469	2.357	0.562
8192	819.764	1.512	1.991	2.990	0.737
32768	3600.854	2.293	2.914	3.667	0.946
131072	–	5.988	5.763	4.969	1.753
524288	–	12.861	11.110	6.442	2.672
2097152	–	28.436	22.106	8.166	3.807
8388608	–	65.205	47.994	9.546	4.889

Table 3: Time per query [µs] for random points on a parabola.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Input Size	CGAL D. hierarchy	ANN kd-tree	CGAL kd-tree	FDH array exact	FDH array inexact
128	7.474	0.623	0.849	0.945	0.196
512	10.784	0.556	0.746	2.011	0.440
2048	14.969	0.650	0.851	3.449	0.603
8192	38.160	0.712	0.904	5.309	1.009
32768	81.072	0.794	0.975	10.787	1.827
131072	294.277	1.068	1.155	14.570	3.900
524288	–	2.084	1.653	32.746	11.473
2097152	–	3.664	2.787	73.713	29.044
8388608	–	6.075	4.642	177.807	76.334

Table 4: Time per query [µs] for for mixed inputs circle / square.

E[#points closer to finger]	exact finger flat	exact finger steep	exact no finger	inexact finger flat	inexact finger steep	inexact no finger
1.6	2.638	2.677	7.829	1.416	1.426	3.084
6.3	3.100	3.238	7.810	1.713	1.766	3.088
25.2	3.766	4.064	7.931	2.135	2.240	3.104
101	4.680	5.019	7.839	2.713	2.787	3.086
403	5.889	5.995	7.844	3.477	3.338	3.080
1611	7.321	6.932	7.858	4.378	3.846	3.087
6443	8.891	7.816	7.877	5.344	4.268	3.080
25771	10.455	8.552	7.909	6.281	4.551	3.075
103085	12.060	9.139	7.834	7.063	4.654	3.024
412339	12.986	9.625	7.802	7.460	4.604	2.919
1649358	13.383	9.659	7.585	7.226	4.317	2.658

Table 5: Time per query [µs] of finger queries on 2^{23} random points in a square.

Number of Points	FDH arr exact	FDH arr inexact + FDH arr exact finger	FDH arr inexact
16	0.492	1.225	0.087
64	0.911	0.695	0.151
256	1.396	0.899	0.207
1024	1.768	1.020	0.291
4096	2.238	1.156	0.375
16384	2.732	1.675	0.505
65536	3.739	2.509	1.018
262144	4.683	3.224	1.579
1048576	5.700	3.957	2.157
4194304	6.716	4.758	2.775

Table 6: Time per query [µs] when using finger search to speedup exact non-finger search for random points in a square.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Input Size	ANN kd-tree	CGAL D. h.	CGAL kd-tree	FDH array	FDH
128	0.445	1.422	0.438	1.977	1.859
512	0.424	1.256	0.408	1.936	1.699
2048	0.461	1.229	0.418	1.980	1.675
8192	0.505	1.231	0.432	2.008	1.708
32768	0.555	1.253	0.478	2.356	1.814
131072	0.714	1.308	0.538	2.608	1.922
524288	1.244	1.375	0.710	2.727	2.044
2097152	1.748	1.457	0.897	2.824	2.129
8388608	2.316	1.520	1.084	2.889	2.193

Table 7: Construction [µs] time per random point in a square.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.