Lazy Algorithms for Dynamic Closest Pair with Arbitrary Distance Measures

Jean Cardinal* David Eppstein†

Abstract

We propose novel lazy algorithms for the dynamic closest pair problem with arbitrary distance measures. In this problem we have to maintain the closest pair of points under insertion and deletion operations, where the distance between two points must be symmetric and take value in a totally ordered set. Many geometric special cases of this problem are well-studied, but only few algorithms are known when the distance measure is arbitrary. The proposed algorithms use a simple delayed computation mechanism to spare distance calculations, and one of these algorithms is a lazy version of the FastPair algorithm recently proposed by Eppstein. Experimental results on a wide number of applications show that lazy FastPair performs significantly better than FastPair and the other algorithms we tested.

1 Introduction

We study the dynamic closest pair problem: given a distance measure between any two points, maintain the closest pair of a set of points under insertion and deletion operations. This is a classical topic, many geometrical special cases of which have been extensively studied before [3, 4].

In this paper, we study the problem with as few assumptions as possible. The only conditions imposed on the distance measure $D(.,.)$ between two points are that it must be symmetric and take values in a totally ordered set. We avoid relying on any property of the distance such as Euclideanness or the triangle inequality.

This problem has a wide range of applications, a good survey of which can be found in Eppstein's paper [8]. Among others is agglomerative clustering, a well-known clustering technique that iteratively merges the two closest clusters [7]. The algorithm starts with as many clusters as points, and ends as soon as the number of clusters is sufficient, or when some threshold in the objective value is reached. This family of algorithms can make use of dynamic closest pair subroutines, a merging corresponding to two deletions and one insertion. Cluster similarity can be measured in several ways, and such routines should not rely on any assumptions about the similarity measure. It is worth noticing that in the vector quantization literature, this algorithm is known as the Pairwise Nearest Neighbor (PNN) method [10]. An example of PNN-like algorithm that uses a more sophisticated distance measure can be found in [6]. The PNN method is a good example of a practical application that uses distance measures which have no useful geometric properties. It is shown in [8] that most known softwares implementing these algorithms use a brute force search method to find the optimal mergings. Other applications are greedy matching in graphs, traveling salesman heuristics and ray-intersection diagrams.

Efficient practical methods for this problem have been presented in [8], derived from the algorithms described earlier in [9] for various geometric problems. Implementations and links can be found on Eppstein's website[1]. We propose improvements of these methods based on lazy deletion, in which actual distance recomputations are are delayed until a closest pair query requires them.

Section 2 presents previously proposed algorithms for this problem. Section 3 presents new algorithms using a lazy deletion mechanism. In section 4 we comment on several experimental results. Conclusion and directions for future work are found in section 5. This manuscript is a revised version of a preliminary technical report [5].

2 Previous Algorithms

The following algorithms are all described in [8], although the first one is a simple method that can be found in other works. Conga lines and Multiconga are the only known linear-space algorithms providing non-trivial amortized bounds on the update complexities. Theoretical lower bounds on the complexity of the update operations are not known.

*Université Libre de Bruxelles, Brussels, Belgium, jcardin@ulb.ac.be

†University of California, Irvine, USA, eppstein@ics.uci.edu

[1] `http://www.ics.uci.edu/~eppstein/projects/pairs/`

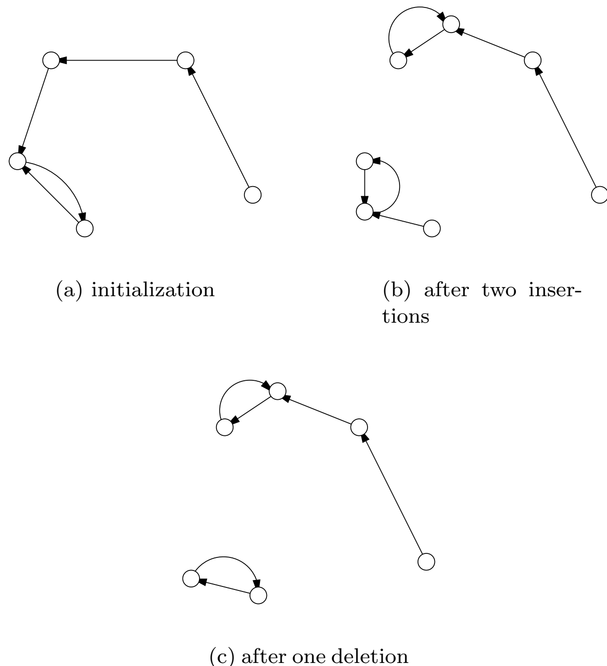


Figure 1: Illustration of the neighbor heuristic.

2.1 Neighbor Heuristic. The neighbor heuristic associates with each point p of S its nearest neighbor in $S \setminus \{p\}$, together with the corresponding distance $d(p) = \min_{q \in S \setminus \{p\}} D(p, q)$.

These data are maintained for each operation. A query consists in scanning all the distances $d(p)$ and selecting the smallest. Worst-case time complexities are $O(n)$, $O(n)$ and $O(n^2)$ for query, insertion and deletion, respectively. This algorithm is actually a simple maintenance of the nearest neighbor directed graph structure. It is illustrated in Fig. 1.

2.2 Conga Lines and MultiConga. A *conga line* for a subset S_i of the data points is a directed path, found by choosing a starting point arbitrarily, and then selecting each successive point to be the nearest unchosen neighbor of the previously selected point. Points within S_i may choose their neighbors from the entire data set, while points outside S_i must choose their neighbors from within S_i. The *conga lines data structure* consists of a partition of the points into $O(\log n)$ sets S_i, and a graph G_i for each set, with G_i initially constructed to be a conga line for S_i although subsequent deletions may degrade its structure. The data structure also contains a priority queue of the edges in the union of the graphs G_i, from which a closest pair can be found as the shortest edge. Each insertion creates a new set S_i, and each deletion causes the vertices connected by graph edges to the deleted point to be moved to a new set S_i; after every such update some sets are merged and their graphs reconstructed in order to maintain the $O(\log n)$ bound on the number of sets.

THEOREM 1. (CONGA LINES [8]) *Conga Lines correctly maintain the closest pair in amortized time $O(n \log n)$ per insertion and $O(n \log^2 n)$ per deletion.*

Proof. We define $n^2 \log n - n \sum_{i=1}^{k} |S_i| \log |S_i|$ as potential function, and study the sum of the effective cost of the operation and the corresponding potential function variation. □

MultiConga is similar to Conga Lines, except that subsets S_i are never merged: the number of subsets is allowed to become arbitrarily large.

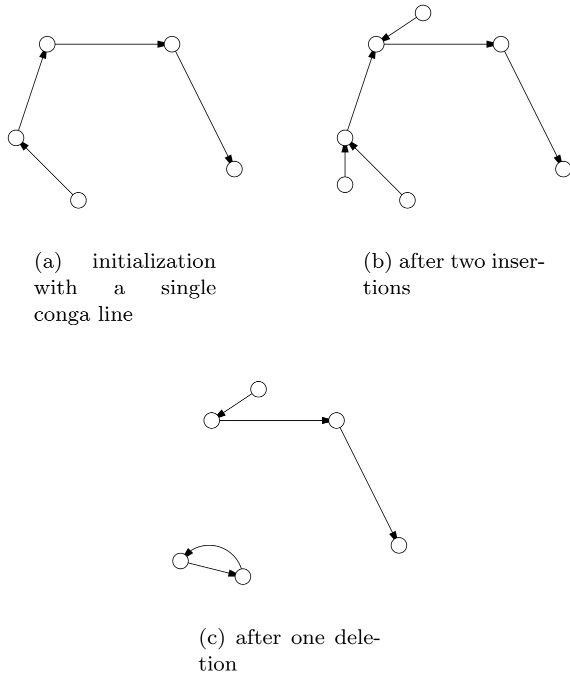
THEOREM 2. (MULTICONGA [8]) *MultiConga correctly maintains the closest pair in amortized time $O(n)$ per insertion and $O(n^{3/2})$ per deletion.*

2.3 FastPair. FastPair can be seen as a further relaxation of MultiConga, but is better explained in terms of the neighbor heuristic. In this method, when a new point is inserted, its nearest neighbor is computed, but nearest neighbors and distances $d(y)$ associated with other points y in the set are not updated. Hence the points that were previously in the set may be associated with wrong neighbors. This is not important since the correct distance can still be found associated with the new point. A second difference with the neighbor heuristic is that the structure is initialized with a single conga line, instead of computing all possible distances. In terms of complexity bounds, FastPair does not improve on the neighbor heuristic. However, experimentally, it was shown to be the best overall choice [8].

If we refer to conga lines, FastPair is a MultiConga data structure in which vertices that are moved after a deletion do not form a new subset, but rather a collection of singletons. It is illustrated on Fig. 2.

3 Lazy Algorithms

We present new algorithms that extend the previous ones by using a lazy deletion procedure. While the basic idea is quite simple, it is not immediately clear that it is applicable in all situations, in particular when the distance is not geometric, and in combination with the FastPair algorithm. We show that it is indeed the case and, in section 4, that it provides what can be considered as the fastest known practical algorithm for generic dynamic closest pair maintenance.



(a) initialization with a single conga line

(b) after two insertions

(c) after one deletion

Figure 2: Illustration of the FastPair algorithm.

3.1 Lazy Neighbor Heuristic. Lazy deletion spares a certain amount of distance recalculations by delaying them as most as possible. It works as follows: when a point p is deleted, we do not recompute the nearest neighbors of the points that were having p as nearest neighbor. Instead, we simply mark them. When a query is made, the smallest distance $d(p)$ is selected using a linear search. If p is marked, we recompute its nearest neighbor and iterate until the smallest distance corresponds to a nonmarked point, which must belong to the closest pair. When a point p is inserted we search for its nearest neighbor, and at the same time modify nearest neighbors of the previously existing points that have p as new nearest neighbor. If some of these points were marked, we unmark them, since the distance is again valid.

Lazy deletion was also proposed by Kaukoranta et al. [11] to accelerate the PNN algorithm. They restrict the use of lazy deletion to distances satisfying a so-called monotony criterion, and show that the weighted distance used in the PNN algorithm satisfies it. It is interesting to note that it can actually be implemented for any type of symmetric, totally ordered distance. The main difference between the lazy neighbor heuristic and the algorithm in [11] is that in our method some points may be unmarked during an insertion.

Theorem 3. (Correctness) *The lazy neighbor heuristic correctly maintains the closest pair*

Proof. Any distance $d(p)$ associated with a marked point p is less than or equal to its actual nearest neighbor distance. The correctness of the algorithm follows. □

In terms of the number of distance calculations, the lazy neighbor heuristic cannot cost more than the standard neighbor heuristic. It is not difficult, however, to show that amortized time bounds are the same as in the unmodified neighbor heuristic, i.e. equal to the worst-case bounds.

3.2 Lazy FastPair. Lazy FastPair combines the lazy insertion algorithm of FastPair with lazy deletion. Hence for an insertion we compute the nearest neighbor of the new point, but do not change previously computed neighbors, while for a deletion we mark certain points as having deleted neighbors. If, in a query, it happens that $d(p)$ is the minimum distance and p is marked, we recompute the nearest neighbor of p and iterate. For this algorithm, it may happen that the actual nearest neighbor distance of a marked point is smaller than the stored distance. This does not harm the correctness of the algorithm, as shown in the following.

Theorem 4. (Correctness) *Lazy FastPair correctly maintains the closest pair.*

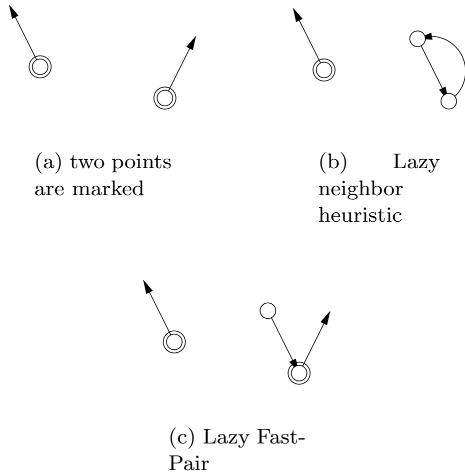
Proof. All the operations maintain the following invariant: Let p be a point in S such that $d(p)$ is minimal; then either there exists $q \in S$ such that (p, q) is the closest pair, or p is marked.

Suppose, on the contrary, that p is not marked, $d(p)$ is minimal, but (a, b) is the closest pair, with $a \neq p$, $b \neq p$ and $D(a, b) < d(p)$. Without loss of generality, suppose also that b was inserted after a. Then b must be marked, otherwise we would have $d(b) = D(a, b) < d(p)$. And when b was inserted, its associated distance must have been at most equal to $D(a, b)$. Hence $d(p)$ cannot be smaller than $d(b)$. So (a, b) is not the closest pair and p must belong to the closest pair.

From this invariant, it is easy to check that the query procedure returns the closest pair. □

It is interesting to notice that the invariant given in the proof of correctness for the lazy neighbor heuristic is not a necessary condition. Also, in lazy FastPair, a point cannot be unmarked during an insertion step. This is illustrated on Fig. 3.

As for the lazy neighbor heuristic, amortized time bounds do not change. It is easy to find a sequence of worst cases in which the algorithm does not perform



(a) two points are marked

(b) Lazy neighbor heuristic

(c) Lazy FastPair

Figure 3: Illustration of the insertion procedure : in Lazy FastPair, an insertion cannot unmark a point.

better than the neighbor heuristic. This sequence occurs when at some point the nearest neighbor graph is star-shaped and the point at the center of the star is deleted. Then all remaining points are marked. Subsequent queries might be forced to find the nearest neighbors of all these marked points, and eventually reconstruct a star-shaped graph.

3.3 Lazy Conga Lines. The question naturally arises of whether it is possible to combine lazy deletion with the conga lines data structure, and whether this can lead to better complexity bounds.

An outline of algorithm is as follows. We maintain $O(\log n)$ conga lines as before. When inserting a point, we create a new conga line for it. During deletion, the point to be deleted is removed, and points that were pointing to it are marked. When a query is made, we check the top of the heap that is used to store conga line edges. We iteratively extract edges from the heap until an edge corresponding to an unmarked point is found. A new subset S_i is created for all the extracted edges, a new conga line is created, and the graphs are merged until the number of subsets is $O(\log n)$. At that point, an unmarked edge can be found at the top of the heap, that corresponds to the actual closest pair.

Correctness of the algorithm follows from correctness of the conga lines data structure and that of lazy deletion. Essentially, the algorithm is similar to lazy FastPair, except that when the query procedure finds a marked point, it is not reinserted immediately, but is inserted together with all the other marked points at the top of the heap. Another difference is the maintenance of $O(\log n)$ subsets, which is not ensured in FastPair. At most $O(\log n)$ points are marked at each deletion (one from each subset), and all these points might later be reinserted in amortized time $O(n \log n)$, as in the original conga lines data structure. Hence the amortized time of deletion remains $O(n \log^2 n)$.

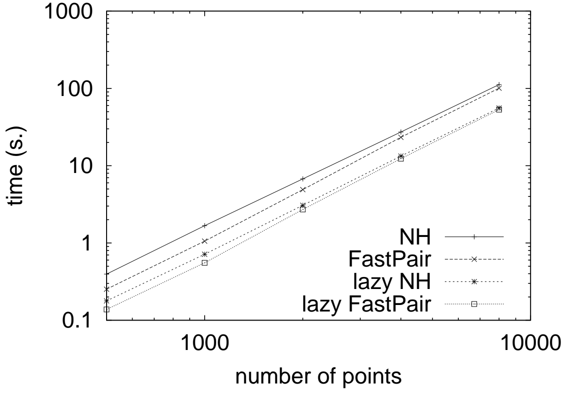
4 Experiments

We implemented the lazy neighbor heuristic and lazy FastPair and ran them on the applications from Eppstein's testbed [8]. This includes agglomerative clustering, as described in the introduction, as well as greedy matching in graphs as described by Reingold and Tarjan [12] and the multifragment heuristic from Bentley [1, 2]. We refer the reader to the reference paper for further details about the testbed. We did not implement lazy conga lines structure, because we did not find any rationale why this could be faster than Lazy FastPair in general. This implementation could however be planned later for completeness, especially in a maximum weight matching application previously identified as the only application for which conga lines performed better than FastPair. We did not compare our new algorithms to the quadtree method from the same paper, that requires quadratic storage complexity. We assume that the number of points is high enough so that linear storage complexity is a necessary condition.

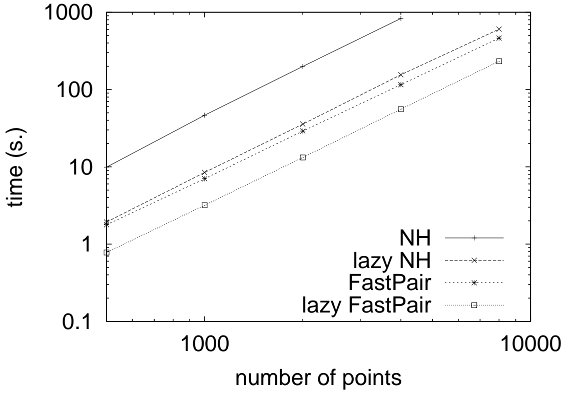
In Fig.4–7 we report the execution times in seconds with respect to the number of points. For readability, and to ease the comparison with the results in [8], the plots are presented with logarithmic scales. The machine used was a Pentium IV, 1.7 GHz, running Linux. Note that the FastPair and neighbor heuristic methods never perform less distance calculations than their lazy counterparts, hence having compared the number of distance calculations would have been beneficial for the lazy methods.

The lazy neighbor heuristic performed better than FastPair on most agglomerative clustering tasks – except when using the L_∞ distance – but significantly worse than it on the multifragment TSP heuristic application. Lazy FastPair performed better than all other algorithms, except on agglomerative clustering in a fractal set, for which the lazy neighbor heuristic is better when the number of points is greater than 1500. Lazy FastPair performed strikingly better than all algorithms on the clustering application with the L_∞ distance.

In general, it can be seen that these algorithms run in near quadratic time, which in our experiments means that updates are made in near linear time. This can be verified on Fig. 8 showing three graphs in which the running time is plotted with respect to n^2. Quadratic behavior was already observed in [8] and explains why algorithms that have better worst-case amortized time

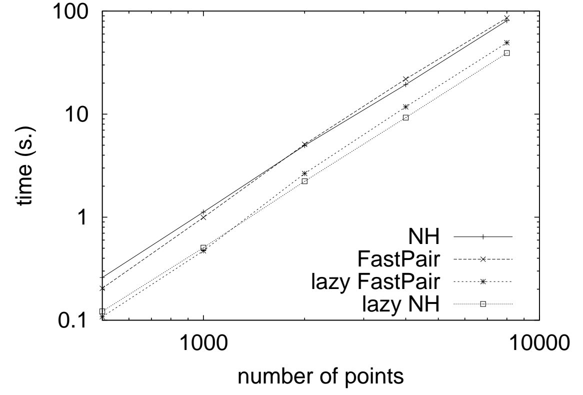


(a) clustering of uniformly distributed points in the 20-dimensional hypercube with the L_1 metric

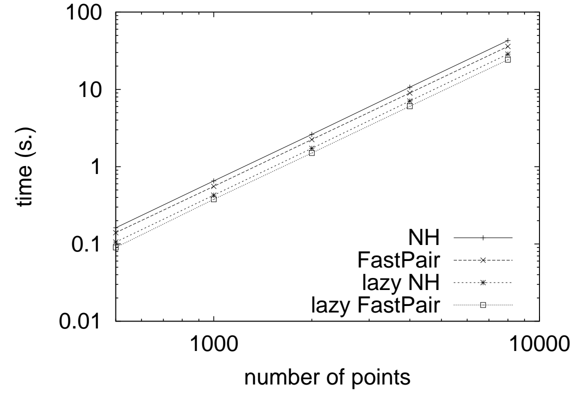


(b) clustering of uniformly distributed points in the 31-dimensional hypercube with the L_∞ metric

Figure 4: algorithms performance for hierarchical clustering with L_p metrics

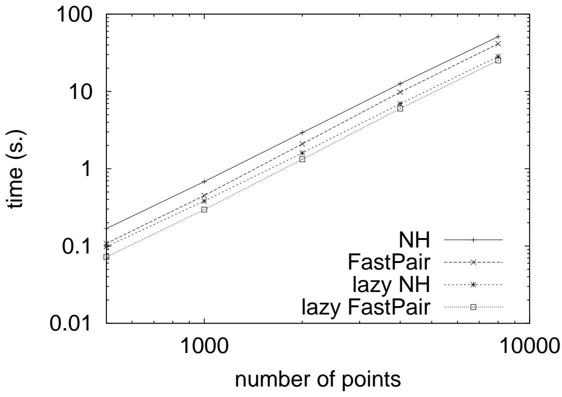


(a) clustering of points in a 31-dimensional Sierpinski tetrahedron

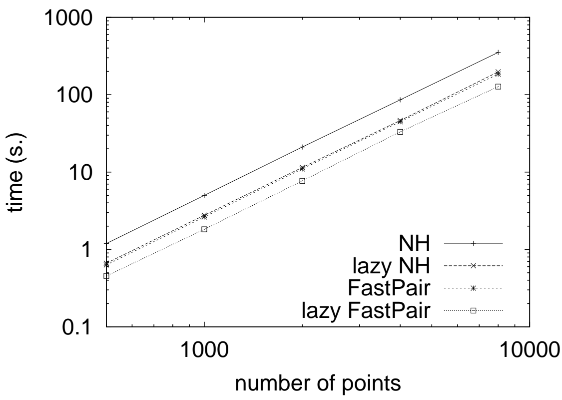


(b) clustering with pseudorandom distance matrix

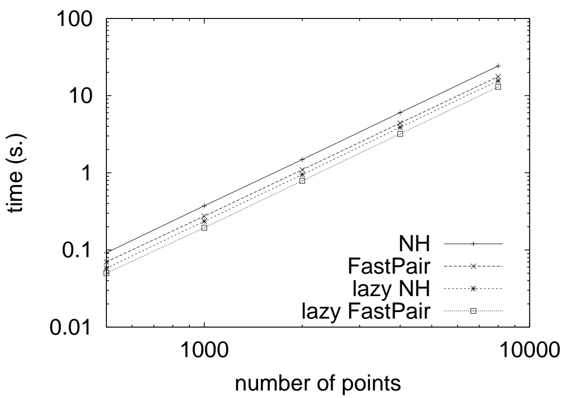
Figure 5: algorithms performance for hierarchical clustering with other metrics



(a) greedy minimum matching of points in the 20-dimensional hypercube with the L_1 metric

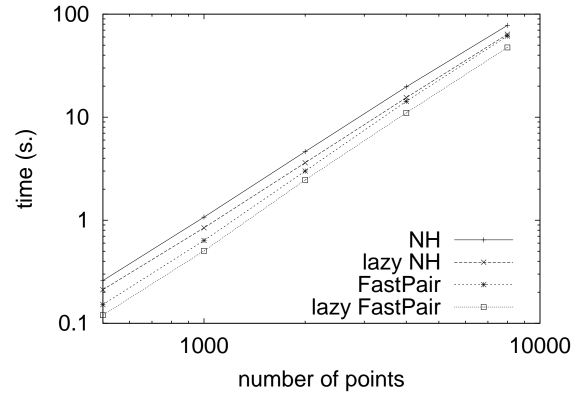


(b) greedy minimum matching of points in the 31-dimensional hypercube with the L_∞ metric

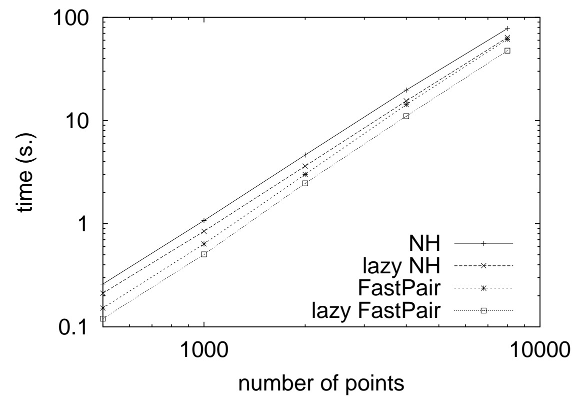


(c) greedy minimum matching with pseudorandom distance matrix

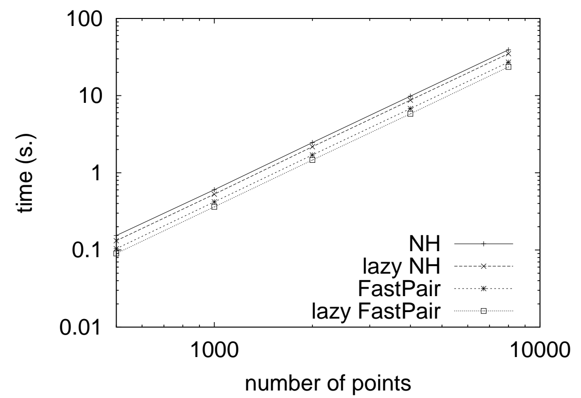
Figure 6: algorithms performance for greedy minimum matching



(a) multifragment TSP heuristic in the 20-dimensional hypercube with the L_1 metric



(b) multifragment TSP heuristic in the 31-dimensional hypercube with the L_∞ metric



(c) multifragment TSP heuristic with pseudorandom distance matrix

Figure 7: algorithms performance for the multifragment TSP heuristic

bounds such as conga lines are not winners in the comparison: the worst case does not seem to be relevant in any of the experiments.

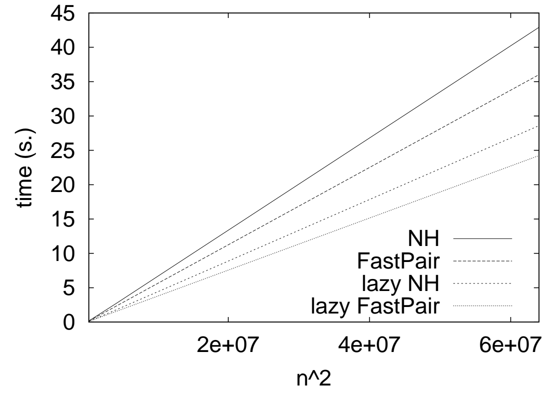
A quadratic complexity implies that the average number of nearest-neighbor searches per iteration is a constant number. In order to verify this, we measured the overall number of nearest neighbor searches and divided it by the number of closest pair queries. The results are presented in Table 1 for the three applications with pseudorandom distances using Lazy FastPair. For the greedy matching and multifragment heuristic applications, this number is equal to the average number of iterations performed in the closest pair query before an unmarked point is found. It is greater than one in the clustering case, since an insertion, that always costs one search, is performed at each iteration to include the newly formed cluster in the set. We can see that the number can be considered as constant in the three cases for pseudorandom distances. The results are similar for the other distances: with the L_∞ metric in 31 dimensions, for instance, the number of searches per closest pair query is about 1.17 for the clustering, 0.68 for the greedy matching and 0.93 for the multifragment TSP heuristic.

5 Conclusion

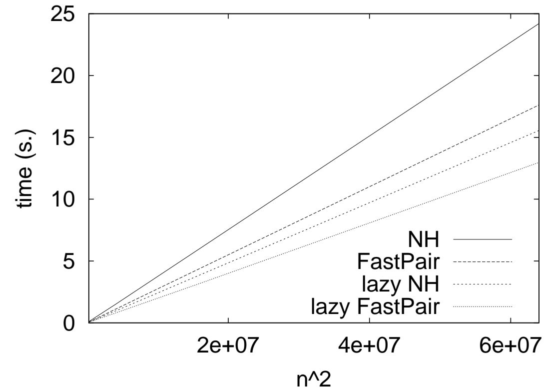
We proposed two simple lazy algorithms for the dynamic closest pair problem with arbitrary distance measure. Lazy FastPair performs better than all previously proposed algorithm, although it has the same worst-case and amortized bounds as the simple neighbor heuristic. These algorithms are applicable to many different problems and can lead to much more efficient applications in practice, especially when distance calculation costs are high. Experiments should be carried out, for instance, on string clustering tasks using the edit distance, whose calculation requires a dynamic programming subroutine. We recommend the use of the Lazy FastPair algorithm in all situations where linear space complexity is required.

Among other issues, we may wonder whether it is possible to make these algorithms even more lazy, or use laziness in another way, and whether it is possible to use laziness to obtain better amortized complexity bounds. The prediction of the average acceleration ratios on some known distributions might also be of interest. Finally, the most interesting theoretical issue is probably that of the lower bounds on the complexity of each operation.

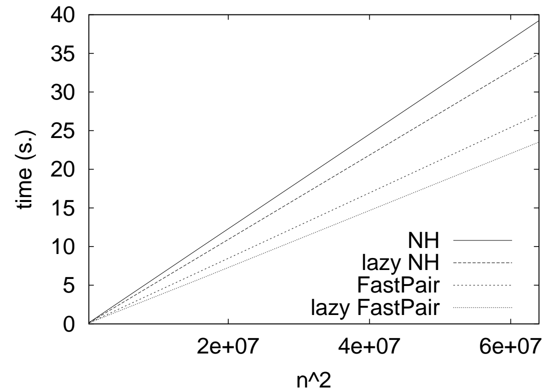
Acknowledgements. The authors thank the anonymous reviewer for his suggestions and for pointing out reference [2].



(a) clustering with pseudorandom distance matrix



(b) greedy minimum matching with pseudorandom distance matrix



(c) multifragment TSP heuristic with pseudorandom distance matrix

Figure 8: plots of the running time vs. n^2

number of points	average number of searches
500	1.429
1000	1.424
2000	1.455
4000	1.457
8000	1.451

(a) clustering

number of points	average number of searches
500	0.537
1000	0.644
2000	0.629
4000	0.625
8000	0.637

(b) greedy minimum matching

number of points	average number of searches
500	0.845
1000	0.831
2000	0.838
4000	0.840
8000	0.839

(c) multifragment TSP heuristic

Table 1: average number of nearest-neighbor searches per closest pair query

References

[1] J. Bentley. Experiments on traveling salesman heuristics. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 91–99, 1990.

[2] ———. Fast Algorithms for Geometric Traveling Salesman Problems. In *ORSA Journal on Computing*, 4(4):387–411, 1992.

[3] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 1995.

[4] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[5] J. Cardinal and D. Eppstein. Lazy algorithms for dynamic closest pair with arbitrary distance measures. Technical Report 502, ULB, 2003.

[6] D. P. deGarrido, W. A. Pearlman, and W. A. Finamore. A clustering algorithm for entropy-constrained quantizer design with applications in coding image pyramids. *IEEE Trans. on Circuits and Sytems for Video Technology*, 5:83–85, 1995.

[7] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[8] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Experimental Algorithmics*, 5(1):1–23, 2000. also in 9th ACM-SIAM Symp. on Discrete Algorithms, 1998, pp. 619-628.

[9] D. Eppstein, P. Agarwal, and J. Matousek. Dynamic algorithms for half-space reporting, proximity problems, and geometric minimum spanning trees. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1992.

[10] W. H. Equitz. A new vector quantization clustering algorithm. *IEEE Trans. Acoust., Speech, Signal Processing*, 37(10):1568–1575, October 1989.

[11] T. Kaukoranta, P. Franti, and O. Nevalainen. Fast and space efficient PNN algorithm with delayed distance calculations. In *8th International Conference on Computer Graphics and Visualization (GraphiCon'98)*, 1998.

[12] E. M. Reingold and R. E. Tarjan. On a greedy heuristic for complete matching. *SIAM J. of Computing*, 10(4):676–681, 1981.