Efficient Disk Set Matching Under Noise

Yago Diez * J. Antoni Sellarès *

*Institut d'Informàtica i Aplicacions, Universitat de Girona, Spain. Email: {ydiez,sellares}@ima.udg.es. Partially supported by the Spanish Ministerio de Educación y Ciencia under grant TIN2004-08065-C02-02.

Abstract

Let $\mathcal{A}$ and $\mathcal{B}$ be two disk sets, with $|\mathcal{A}| \leq |\mathcal{B}|$. We propose a process for determining matches between $\mathcal{A}$ and subsets of $\mathcal{B}$ under rigid motion, assuming that the position of all disks in both sets contains a certain amount of "noise". The process consists of two main stages: a candidate zone determination algorithm and a matching algorithm. A candidate zone is a region that contains a subset $\mathcal{S}$ of $\mathcal{B}$ such that $\mathcal{A}$ may match one or more subsets $\mathcal{B}'$ of $\mathcal{S}$. We use a compressed quadtree to have easy access to the subsets of $\mathcal{B}$ related to candidate zones. In each quadtree node we store geometric information that is used by the algorithm that searches for candidate zones. The second algorithm solves the disk set matching problem: we generate all, up to a certain equivalence, possible motions that bring $\mathcal{A}$ close to some subset $\mathcal{B}'$ of every $\mathcal{S}$ and seek for a matching between sets $\mathcal{A}$ and $\mathcal{B}'$. To find all these possible matchings we use a bipartite matching algorithm that uses Skip Quadtrees for neighborhood queries. We have implemented the proposed algorithms and we report results that show their good performance in practice.

1 Introduction

Determination of the presence of a geometric pattern in a large set of objects is a fundamental problem in computational geometry. We encounter matching problems in fields such as machine vision, astronautics, document processing, computational biology and computational chemistry. In some cases, as in the constellation recognition problem (considering fixed magnification) in astronautics or the substructure search problem in molecular biology, the objects to be matched are modeled as disks in the plane or balls in space. Stars can be seen as disks with radii determined by their brightness and an atom in a protein molecule can be modeled as a ball whose radius is the Van Der Waals radius of the element it represents. In both cases only a discrete range of radius is considered. Since star and atom positions are fuzzy, both problems can be transformed to approximate disk/ball set matching under rigid motion problems. In this paper we will concentrate in the two-dimensional case.

1.1 Problem formulation

Fixed a real number $\epsilon \geq 0$, we say that two disks $D(a, r)$, $D(b, s)$ *approximately match* when $r = s$ and $d(a, b) \leq \epsilon$, where d denotes the Euclidean distance.

Let $\mathcal{D}$, $\mathcal{S}$ be two disk sets of the same cardinality. A *radius preserving bijective mapping* $f : \mathcal{D} \to \mathcal{S}$ maps each disk $A = D(a, r) \in \mathcal{D}$ to a distinct and unique disk $f(A) = D(b, s) \in \mathcal{S}$ so that $r = s$. Let $\mathcal{F}$ be the set of all radius preserving bijective mappings between $\mathcal{D}$ and $\mathcal{S}$. The *bottleneck distance* between $\mathcal{D}$ and $\mathcal{S}$ is defined as:

$$d_b(\mathcal{D}, \mathcal{S}) = \min_{f \in \mathcal{F}} \max_{A \in \mathcal{D}} d(A, f(A)) .$$

The **Noisy Disk Matching (NDM)** problem can be formulated as follows. Given two disk sets $\mathcal{A}$, $\mathcal{B}$, $|\mathcal{A}| = n$, $|\mathcal{B}| = m$, $n \leq m$, and $\epsilon \geq 0$, determine all rigid motions τ for which there exists a subset $\mathcal{B}'$ of $\mathcal{B}$ such that $d_b(\tau(\mathcal{A}), \mathcal{B}') \leq \epsilon$.

If τ is a solution to the **NDM** problem, every disk of $\tau(\mathcal{A})$ approximately matches to a distinct and unique disk of $\mathcal{B}'$ of the same radius, and we say that $\mathcal{A}$ and the subset $\mathcal{B}'$ of $\mathcal{S}$ *approximately match* or are *noisy congruent*.

If we think of a point as a disk of 0 radius and point sets of the same cardinality are considered, then the **NDM** problem becomes the **Noisy Matching (NM)** problem: Given two point sets $\mathcal{A}$, $\mathcal{B}$ of the same cardinality and $\epsilon \geq 0$, determine, if possible, a rigid motion τ such that $d_b(\tau(\mathcal{A}), \mathcal{B}) \leq \epsilon$.

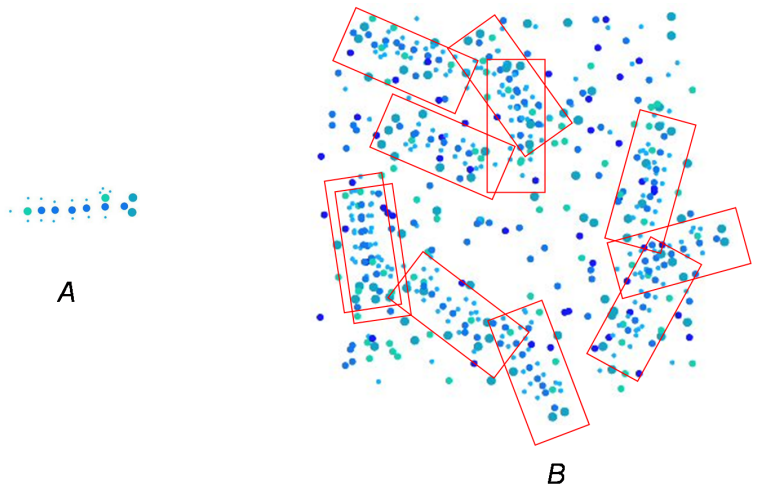

Figure 1: *Our problem consists on finding all the subsets of $\mathcal{B}$ that approximately match to some rigid motion of set $\mathcal{A}$. Red rectangles contain such subsets.*

1.2 Previous Results

The study of the **NM** problem was initiated by Alt *et al.* [1] who presented an exact $O(n^8)$ time algorithm for solving the problem for two sets $\mathcal{A}, \mathcal{B}$ of cardinality n. This bound can be reduced to $O(n^3 \log n)$ if an assignment of points in $\mathcal{A}$ to points in $\mathcal{B}$ is given [1]. Combining Alt et alt. algorithm with the techniques by Efrat *et al.* [5] the time can be reduced to $O(n^7 \log n)$.

To obtain faster and more practical algorithms, several authors proposed algorithms for restricted cases or resorted to approximations [8, 5]. This line of research was initiated by Heffernan and Schirra [8] who presented an $O(n^{2.5})$-time algorithm conditioned to the fact that the noise regions were small. Indyk and Venkatasubramanian [10] claim that this last condition can be removed without increasing the computational complexity using the techniques by Efrat, Itai and Katz [5]. We must note here that this type of approximation refers not to the position of points but to the total approximation achieved, meaning that a better approximations than the output of the algorithm may exist and that the possible error can be bounded according to the parameters of the problem. We will not be interested in this second type of approximation and will demand that our solutions be exact in this sense. Indyk and Venkatasubramanian also reduce the geometric pattern matching problem to a combinatorial one using Hall's theorem and the *generalized bottleneck distance*, which includes the bottleneck distance as a particular case.

2 Our Approach

The main idea in our algorithm is to discretize the **NMD** problem by turning it into a series of "smaller" instances, whose combined solution is faster than the original problem's. To achieve this discretization, we use a conservative strategy that discards those subsets of $\mathcal{B}$ where no noisy match may happen and keep a number of zones where this matches may occur.

We assume that all rectangles and squares we consider are axis-parallel. Our algorithm consists of two main parts. The first one yields a collection of *candidate zones*, which are regions determined by one, two or four squares that contain a subset $\mathcal{S}$ of $\mathcal{B}$ such that $\mathcal{A}$ may approximately match one or more subsets $\mathcal{B}'$ of $\mathcal{S}$. The second part of the algorithm solves the **NDM** problem between $\mathcal{A}$ and every $\mathcal{B}'$.

The discarding decisions throughout the first part of this process are made according to a series of geometric parameters that are invariant under rigid motion. These parameters help us to

describe the shapes of $\mathcal{A}$ and the different subsets of $\mathcal{B}$ that we explore. To navigate $\mathcal{B}$ and have easy access to its subsets, we use a *compressed quadtree* [2]. By doing this we achieve a reduction of the total computational time, corresponding to a pruning of the search space, as an effect of all the calculations we avoid by discarding parts of $\mathcal{B}$ cheaply and at an early stage.

The candidate zone determination algorithm consists itself on two subparts. A quadtree construction algorithm and a search algorithm that traverses the quadtree looking for the candidate zones. The quadtree construction algorithm can also be subdivided in two more parts: a compressed quadtree building algorithm that uses the centers of the disks in $\mathcal{B}$ as sites (without considering their radii), and an algorithm that adds the information related to the geometric parameters being used to each node.

The second part of the algorithm consists on two more parts. The first one, the "enumeration" part, groups all possible rigid motions of $\mathcal{A}$ in equivalence classes in order to make their handling feasible. We then choose a representative motion τ for every equivalence class. The second step, the "testing" part, performs a bipartite matching algorithm between every set $\tau(\mathcal{A})$ and every disk set $\mathcal{B}'$ associated to a candidate zone. For these matching tests we modify the algorithm proposed in [5] by using the *skip-quadtree* data structure [6] in order to make it easier to implement and to take advantage of the data structures that we have already built.

Our contribution can be summarized as follows::

- We address new aspects of the NM problem generally not taken into account, such as considering the radii of the disks or the fact that the two sets involved may not have the same cardinality and, thus, yield more than one possible matching.
- Although our algorithms are designed to solve the generic NDM problem, the possibility to define different geometric parameters allows the algorithm to take advantage of the characteristic properties of specific applications.
- We have implemented the algorithm presented, which represents a major difference to the previous and mainly theorical approaches. The implementations rely on widely used data structures what enhances its applicability.
- As will be discussed in the results Section, the performance of the algorithm depends on the effectiveness that the pruning step and the different parameters have on every data set, but at its worst, when the pruning step has no effect at all and all the disks have the same radius, it meets the best (theoretical) running time up to date. Assuming a "perfect" behavior of the pruning step, the initial problem is transformed to a series of problems of the same kind but with cardinality $n = |\mathcal{A}|$, producing a great saving of computational effort.
- Our methods are parallelizable, not only because calculations in its search step that run on different subsets of the compressed quadtree can take place simultaneously, but also because the subproblems that this search yields are all independent.

3 Candidate zone determination algorithm

Let $R_{\mathcal{A}}$ be the minimal rectangle that contains all the centers of the disks in $\mathcal{A}$, and let s be the smallest positive integer for which $(\text{diagonal}(R_{\mathcal{A}}) + 2\epsilon) \leq 2^s$ holds. It is not difficult to prove that for any rigid motion τ there exists a square of *size* s (with side length 2^s) containing all the centers in $\tau(\mathcal{A})$. This allows us to affirm that, for any $\mathcal{S} \subset \mathcal{B}$ noisy congruent with $\mathcal{A}$ there exists a square of size s that contains the centers of its disks. We store the centers of the disks in $\mathcal{B}$ in a compressed

PR quadtree $\mathcal{Q}_\mathcal{B}$ and describe the geometry of each of the nodes in this quadtree using a number of geometric parameters that are invariant for rigid motions. Then we look for candidate zones in the quadtree whose associated geometric parameters match those of $\mathcal{A}$. To sum up, we can say that, in the first step of the algorithm, instead of looking for all possible rigid motions of set $\mathcal{A}$, we look for squares of size s covering subsets of $\mathcal{B}$, which are parameter compatible with $\mathcal{A}$. Although our intention would be to describe our candidate zones exactly as squares of size s this will not always be possible, so we will also have to use two or four squares of size s. It is important to stress the fact that ours is a conservative algorithm, so we do not so much look for candidate zones as rule out those regions where no candidate zones may appear.

3.1 Compressed quadtree construction

Although for the first part of the algorithm we only use the quadtree levels between the root and the one whose associated nodes have size s, we use the remaining levels later, so we build the whole compressed quadtree $\mathcal{Q}_\mathcal{B}$. We use the techniques in [2] to ensure a total asymptotic cost of $O(m \log m)$ in all cases, where $m = |\mathcal{B}|$.

3.1.1 Adding information to the quadtree

To simplify explanations we consider $\mathcal{Q}_\mathcal{B}$ to be complete. Although it is clear that this is not be the general situation this limitation can be easily overcome in all the parts of the algorithm (see Appendix for details).

At this moment the quadtree $\mathcal{Q}_\mathcal{B}$ contains no information about the different radii of the disks in $\mathcal{B}$ or the geometric characteristics of $\mathcal{B}$ as a whole. Since these parameters will guide our search for matches they must be invariant under rigid motion. The geometric parameters we use are: a) parameters that take into account the fact that we are working with disk sets: number of disks and histogram of disk's radii attached to a node; b) parameters based on distances between centers: maximum and minimum distance between centers. For every geometric parameter we will define a *parameter compatibility criterium* that will allow us to discard zones of the plane that cannot contain a subset $\mathcal{B}'$ of $\mathcal{B}$ to which $\mathcal{A}$ may approximately match (see figure 2 for an example). Other general geometric parameters may be considered in future work as well as specific ones due to specialized applications of the algorithms.

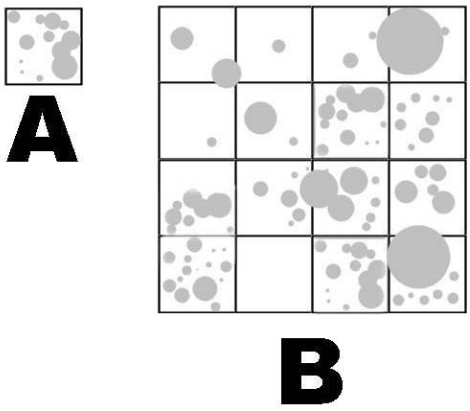

Figure 2: *There cannot be any $\mathcal{B}'$ that approximately matches $\mathcal{A}$ fully contained in the four top-left squares because $\mathcal{A}$ contains twelve disks and the squares only six.*

Once selected the set of geometric parameters to be used, in the second stage of the quadtree construction, we traverse $\mathcal{Q}_\mathcal{B}$ and associate to each node the selected geometric parameters. We also

compute them for the whole of $\mathcal{A}$. The computational cost of adding the geometric information to $\mathcal{Q}_\mathcal{B}$ depends on the parameters that we choose. In the case of the "number of disks" and "histogram of disk's radii" parameters we can easily keep track of them while we build the quadtree, so no additional cost is needed. For the "minimum and maximum distance between centers" parameters, the necessary calculations can be carried out in $O(m \log m)$ time using a divide and conquer approach. Adding other parameters will indeed need extra computational time but will also make the discarding of zones more effective. The balance between this two factors will be an important part of our future work.

3.2 Candidate zone determination

We must face the problem of determining all the candidate zones where squares of size s that cover a subset of $\mathcal{B}$ which is parameter compatible with $\mathcal{A}$ can be located. The subdivision induced by the nodes of size s of $\mathcal{Q}_\mathcal{B}$ corresponds to a grid of squares of size s superimposed to set $\mathcal{B}$. If we bear in mind that we are trying to place a certain square in a grid of squares of the same size, it is easy to see that the only three ways to place one of our squares respect to this grid correspond to the relative position of one of the square's vertices. This yields three different kinds of candidate zones associated to one, two or four nodes (see Figure 3). The subsets $\mathcal{B}'$ that we are looking for may lie anywhere inside those zones.

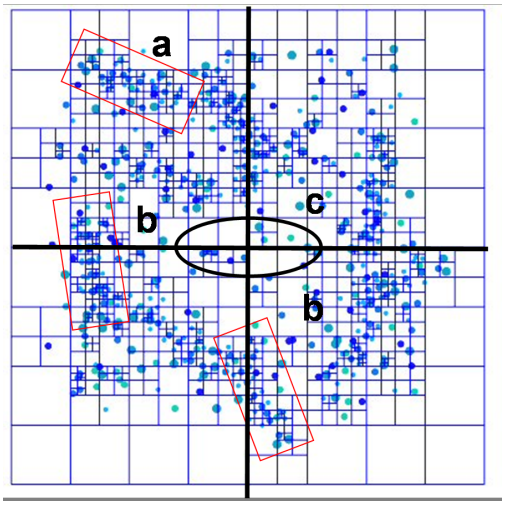

Figure 3: *Position of the candidate zones in the grid. Overlapping: (a) a single grid-square (corresponding to a single quadtree node), (b) two (vertically or horizontally) neighboring nodes, or (c) four neighboring nodes. In this example we observe occurrences of set $\mathcal{A}$ in zones of the first two types and an ellipse showing where occurrences of the third type (not present) would appear.*

3.2.1 Search algorithm

Due to lack of space, we only provide a very brief overview of an algorithm that traverses the quadtree $\mathcal{Q}_\mathcal{B}$ searching for the collection $\mathcal{C}$ of candidate zones (see Appendix for details). The hierarchical decomposition of $\mathcal{B}$ provided by $\mathcal{Q}_\mathcal{B}$ makes it possible to begin searching at the whole of $\mathcal{B}$ and later continue the search only in those zones where, according to the selected geometric parameters, it is really necessary. The algorithm searches recursively in all the quadrants considering also those zones that can be built using parts of more than one of them. The zones taken into account through all the search are easily described in terms of $\mathcal{Q}_\mathcal{B}$'s nodes and continue to decrease their size, until they reach s, following the algorithm's descent of the quadtree. Consequently, early

discards made on behalf of the geometric parameters rule out of the search bigger subsets of $\mathcal{B}$ than later ones.

Given that two or four nodes defining a candidate zone need not be in the same branch of $\mathcal{Q}_\mathcal{B}$, at some points we will need to be exploring two or four branches simultaneously. This will force us to have three separate search functions, depending on the type of candidate zones we are looking for, and to keep geometric information associated to those zones that do not correspond exactly to single nodes in the quadtree but to couples or quartets.

The main search function, denoted search_1, seeks for candidate zones formed by only one node and invokes itself and the other two search functions, called search_2 and search_4 respectively. Consequently, search_2 finds zones formed by pairs of nodes and also launches itself and search_4. Finally, search_4 locates zones formed by quartets of nodes and only invokes itself.

The search step begins with a call to function search_1 with the root node as the parameter. We denote t the size of the root and assume $t >= s$. Function search_1 begins testing if the information in the current node is compatible to the information in $\mathcal{A}$. If this doesn't happen, there is no possible matching contained entirely in the descendants of the current node and we have finished. Otherwise, if the current node has size s then we have found a candidate zone. If this does not happen, we must go down a level on the quadtree. To do so, we consider the four sons of the current node (s_1, s_2, s_3 and s_4).

The candidate zones can be located: **Inside any of the** s_i. So we have to call search_1 recursively in all the s_i's. **Partially overlapping two of the** s_i**'s.** In this case, we would need a function to search both subtrees for all possible pairs of nodes (or quartets) that may arise below in the subdivision. This is function search_2. **Partially overlapping each of the four** s_i**'s.** In this case, we would invoke the function that traverses all four subtrees at a time, called search_4.

Functions search_2 and search_4 work similarly but take into account that they need two and four parameters respectively that those must be chosen adequately. The process goes on recursively until the algorithm reaches the desired size s, yielding a set of candidate zones of all three possible types.

Since in the worst case all possible zones are considered to be candidate zones and the total number of nodes of $\mathcal{Q}_\mathcal{B} \in O(m)$, the number of candidates zones, $c = |\mathcal{C}|$, is in $O(m)$. Considering a "perfect" behavior of the geometric pruning technique and that the quadtree is descended from the root (sized t) down to a level whose nodes have size s, $t - s < m$, then the cost of the search algorithm is $O(c(t-s)) \in O(m^2)$. Summarizing, the computational cost of the search algorithm is $O(\max\{m \log m, c(t-s)\})$. As will be discussed in the Results Section, in our practical experience we find this cost to be much smaller than the cost of the matching algorithm.

4 NDM problem solving algorithm

Now we have a **NDM** problem where the sets involved, $\mathcal{A}$, $\mathcal{S} \in \mathcal{C}$, $n = |\mathcal{A}| \leq n' = |\mathcal{S}| \leq m$, have "similar" cardinality and shape as described by the geometric parameters. Experimental results on this "similarity" will be presented in the Results Section.

We present an algorithm to solve this **NDM** problem, based on the best currently existing algorithms for solving the **NM** problem [1, 5], that takes advantage of the compressed quadtree that we have already built and is implementable. Our approach will consist on two parts called "enumeration" and "testing".

4.1 Enumeration

Generating every possible rigid motion that brings set $\mathcal{A}$ onto a subset of $\mathcal{S}$ is infeasible due to the continuous nature of movement. We partition the set of all rigid motions in equivalence classes in

order to make their handling possible following the algorithm in [1].

For $b \in \mathbb{R}^2$, let $(b)^\epsilon$ denote the circle of radius ϵ centered at point b. Let $\mathcal{S}^\epsilon$ denote the set $\{(b)^\epsilon | D(b,s) \in \mathcal{S}\}$. Consider the arrangement $\mathcal{A}(\mathcal{S}^\epsilon)$ induced by the circles in $\mathcal{S}^\epsilon$. Two rigid motions τ and τ' are considered equivalent if for all disks $D(a,r) \in \mathcal{A}$, $\tau(a)$ and $\tau'(a)$ lie in the same cell of $\mathcal{A}(\mathcal{S}^\epsilon)$. We generate a solution in each equivalence class, when it exists, and its corresponding representative motion using the techniques presented in [1]. A simple geometric argument shows that if there exists any rigid motion τ that solves our **NDM** problem then there exists another rigid motion τ', that belongs to the equivalence class of τ, that also does it and such that we can find two pairs of disks $D(a_i,r_i), D(a_j,r_j) \in \mathcal{A}$ and $D(b_k,s_k), D(b_l,s_l) \in \mathcal{S}$, $r_i = s_k$ and $r_j = s_l$, with $\tau'(a_i) \in (b_k)^\epsilon$ and $\tau'(a_j) \in (b_l)^\epsilon$. We check this property for all quadruples i,j,k,l holding $r_i = s_k$ and $r_j = s_l$. This allows us to rule out those potential matching couples whose radii do not coincide.

Mapping a_i, a_j onto the boundaries of $(b_k)^\epsilon$, $(b_l)^\epsilon$ respectively in general leaves one degree of freedom which is parametrized by the angle $\phi \in [0, 2\pi)$ between the vector $|a_i - b_k|$ and a horizontal line. Considering any other disk $D(a_h,r_h) \in \mathcal{A}$, $h \neq i,j$ for all possible values of ϕ, the center of that disk will trace an algebraic curve σ_{ijklh} of degree six (corresponding to the coupler curve of a four-bar linkage, see [9]), so that for every value of ϕ there exists a rigid motion τ_ϕ holding $\tau_\phi(a_i) \in (b_k)^\epsilon$, $\tau_\phi(a_j) \in (b_l)^\epsilon$ and $\tau_\phi(a_h) = \sigma_{ijklh}(\phi)$. For every remaining disk $D(b_p,s_p)$ in $\mathcal{S}$ with $s_p = r_h$, we compute (using Brent's method for nonlinear root finding, see [3]) the intersections between $(b_p)^\epsilon$ and $\sigma_{ijklh}(\phi)$ which contains at most 12 points. For parameter ϕ, this yields a maximum of 6 intervals contained in $I = [0, 2\pi[$ where the image of $\tau_\phi(a_h)$ belongs to $(b_p)^\epsilon$. We name this set $I_{p,h}$ following the notations in [1]). Notice for all the values $\phi \in I_{p,h}$ we may approximately match both disks. We repeat the process for each possible pair $D(a_h,r_h), D(b_p,s_p)$ and consider the sorted endpoints, called *critical events*, of all the intervals $I_{p,h}$. Notice that the number of critical events is $O(nn')$. Subsequently, any $\phi \in [0, 2\pi[$ that is not one of those endpoints belongs to a certain number of $I_{p,h}$'s and ϕ corresponds to a certain rigid motion τ_ϕ that brings the disks in all the pairs $D(a_h,r_h), D(b_p,s_p)$ near enough to be matched. The subdivision of $[0, 2\pi[$ consisting in all the maximal subintervals that do not have any endpoints of any $I_{p,h}$ in their interior stands for the partition of the set of rigid motions that we were looking for.

In the worst case, $O(n^2 n'^2)$ quadruples of disks are considered. For each quadruple, we work with $O(nn')$ pairs of disks, obtaining $O(nn')$ critical events. Summed over all quadruples the total number of critical events encountered in the course of the algorithm is $O(n^3 n'^3)$.

4.2 Testing

We move parameter ϕ along the resulting subdivision of $[0, 2\pi[$. Every time a critical event is reached, we test the sets $\tau_\phi(\mathcal{A})$ and $\mathcal{S}$ for matching. Whenever the testing part determines a matching of cardinality n we annotate the corresponding τ_ϕ and proceed. Following the techniques presented in [5], in order to update the matching, we need to find a single augmenting path using a layered graph. Each critical event adds or deletes a single edge. In the case of a birth, the matching increases by at most one edge. Therefore, we look for an augmenting path which contains the new edge. If an edge of the matching dies, we need to search for a single augmenting path. Thus in order to update the matching, we need to find a single augmenting path, for which we need only one layered graph.

When searching for augmenting paths we need to perform efficiently two operations. a) **neighbor** $(D(\mathcal{T}), q)$: for a query point q in a data structure $D(\mathcal{T})$ that represents a point set $\mathcal{T}$, return a point in $\mathcal{T}$ whose distance to q is at most ϵ or $\emptyset$ if no such element exists. b) **delete**$(D(\mathcal{T}), s)$: deletes point s from $D(\mathcal{T})$. For our implementation we use the *skip quadtree*, a data structure that

combines the best features of a quadtree and a skip list [6]. The cost of building a skip quadtree for any subset $\mathcal{T}$ of the set of centers of the disks in $\mathcal{S}$ is in $O(n' \log n')$. In the worst case, when $n' = m$, this computational cost is the same needed to build the data structure used in [5]. The asymptotic computational cost of the **delete** operation in $\mathcal{T}$'s skip quadtree is $O(\log n')$. The **Neighbor** operation is used combined with the delete operation to prevent refinding points. This corresponds to a range searching operation in a skip quadtree followed by a set of deletions. The range searching can be approximated in $O(\delta^{-1} \log n' + u)$ time, where u is the size of the output, for a small constant $\delta > 0$ [6]. The approximate range searching outputs some "false" neighbor points that can be detected in $O(1)$ time. We will denote $t(n')$ an upper bound on the amortized time of performing **neighbor** operation in $\mathcal{T}$'s skip quadtree. This yields a computational cost of $O(nt(n'))$ for finding an augmenting path.

Finally for the testing part, since we spent $O(nt(n'))$ time at each critical event for finding an augmenting path, the total time of the algorithm sums $O(n^4 n'^3 t(n'))$.

In the worst case $t(n') \in O(n')$. However, in the most realistic case in which the centers of the disks in $\mathcal{S}$ lie in a somewhat sparse fashion and do not crowd together, there is a constant upper bound on the number of centers in a given disk of radius $(\epsilon + \delta)$ (see [7, 4]) and then $t(n') \in O(\log n')$. In this case our computational cost for finding an augmenting path coincides with the cost obtained in [5] but using simpler data structures, and the total time of the testing algorithm is $O(n^4 n'^3 \log n')$.

4.3 Overall computational costs

If we put together the computational costs of the two parts of the matching algorithm we have:

An $O(\max\{m \log m, c(t-s)\})$ step for the construction of the compressed quadtree with the added geometric information.

Given the cost of $O(n^4 n'^3 t(n'))$ for every candidate zone, when all candidate zones $\mathcal{S} \in \mathcal{C}$ are considered, the total cost is $\sum_{\mathcal{S} \in \mathcal{C}} O(n^4 n'^3 t(n'))$.

As we will see in next Section, even when m is much bigger than n and n', in our practical experiments, this cost tends to be dominated by the cost of the second part.

5 Implementation and results

We have implemented all our algorithm using the C++ programming language under a Linux environment. We used the g++ compiler without compiler optimizations. All tests were run on a Pentium D machine with a 3 Ghz processor. We have carried out a series of synthetic experiments in order to test the performance of our algorithms. We focus specially on the searching algorithm because it contains this paper main contributions.

Before describing the different aspects on which we have focused we state the part that they all have in common. We begin with a data set $\mathcal{A}$ that is introduced by the user. With this data we generate a new set $\mathcal{B}$ that is built applying a (parameterized) number of random transformations (rotations and translation) to set $\mathcal{A}$. At this point, we introduce "white noise" by adding randomly distributed points. The number of "noise points" introduced is $|\mathcal{A}| * (number\_of\_transformations) * (noise\_parameter)$. In order to keep the discussion as simple as possible, all the results in this section refer to an initial set of 25 disks with 6 different type of radii.

Generalities

The first aspect that we want to draw attention to is the fact that the running times of the two algorithms are clearly different. Independently of the sizes of the sets considered the search algorithm is much faster than the matching algorithm, with a difference that increases its value

with the size of set $\mathcal{B}$. Notice that all computational times are given in seconds and that for the sake of clarity we have omitted their non-integer part.

|A|	N.Transf	|B|	Searching Time	Matching Time
25	1	300	0	2
25	5	1500	0	36
25	7	7350	0	54
25	9	9450	0	117
25	12	9600	0	514
25	13	10400	0	532
25	15	12000	0	1194
25	17	13600	0	953
25	19	15200	0	1016
25	21	19425	0	1372

Figure 4: *Compared costs of the Searching and the Matching algorithms*

As we can see in figure 4 even when we have reached a size for set $mathcalB$ that requires quite a lot of computational effort for the matching algorithm, the cost of the search algorithm is still less than a second. To appreciate the cost of the searching step we would have to move up to $|\mathcal{B}| = 35500$ where (the time of the search part is one second) or $|\mathcal{B}| = 247550$ where it is 5 seconds.)

Effects of the Searching Algorithm

We have just seen that the searching algorithm is much faster than the matching algorithm, now we move on to show that it also saves computational time. Figure 5 shows the compared behavior of the matching algorithm undergoing and not undergoing the previous searching algorithm (represented by times T1 and T2 respectively in the figure).

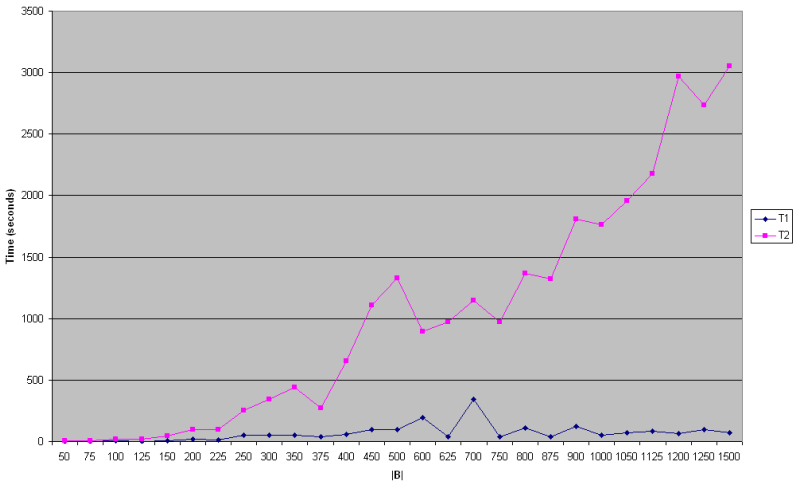

Figure 5: *Algorithm using searching step (T1) and not using it (T2)*

We must state here that the sizes considered here are much smaller given the huge computational time needed by the algorithm that doesn't perform the searching phase. It is clear from the figure that, even when the theorical computational costs are still high, the searching algorithm saves a lot of computational effort.

Experimental Values of n'

We present a brief theoretical discussion on computational times depending on the values of n' and then we will present the practical values that we have obtained in our experiments.

First of all, let us remember that $t(n')$ is in $O(\log n')$ in its best case (due to the delete operation) and in $O(n')$ in its worst. Concerning n', notice that its values are conditioned by the value of n ($n \leq n'$) and by the performance of the searching algorithm. Let's consider a few possible values of n'.

- $n' \simeq n$: In this case the pruning techniques in the searching algorithm would have "worked perfectly" and we would obtain a computational cost of $O(n^7 t(n))$ which behaves like the best algorithms to date and with the enormous advantage that we would be considering only sets of cardinal equal to the smaller of the two sets involved.

- $n' \simeq m$: In this case, the first part of the algorithm cannot have any effect given that there does not exist any possibility to discard any "non-candidate" zone because all set $\mathcal{B}$ is susceptible to be matched to some $\tau(\mathcal{A})$. In this case, the cost is $O(n^4 m^3 t(m))$ and the algorithm behaves similarly to the one in [5] which is not surprising because in this case we would be mainly following it.

- Finally, if we conjecture an intermediate value of n' for example $n'^3 \simeq m$, then we get a computational cost of $O(n^4 m t(m))$ that is, to the best of our knowledge, significantly better than any other of the costs of the algorithms existent up to date. For example, for [5], this cost is $O(n^3 m^4 t(m))$.

|A|	N.Transf	|B|	Mean (n')
25	1	425	25
25	31	13 175	62.37
25	61	25 925	96.40
25	61	48 800	145.84
25	91	72 800	213.27
25	91	106 925	250.78
25	121	96800	271.90
25	121	142 175	328.77
25	151	120800	325.89
25	151	177425	337.87

Figure 6: *Experimental values of n'*

Figure 6 shows some results on the mean of the values of n' for every candidate zone resulting from the execution of the searching algorithm. This tests were carried on with a fixed maximum distance of translation, so as ($|\mathcal{B}|$) grows its disks also get more packed and this produces a higher presence of noisy disks in candidate zones. This is appreciated in the subsequent growth of the mean of n'. However, it is clear that n' stays far from m for all the values presented and close enough to n to fit in one of the other two previous supositions for the majority of them.

6 Future work

In our future work we will study the effect of considering other "geometric parameters" and δ values (used in the approximate range queries) in the efficiency of the algorithm. We also aim at using parallelization techniques to improve the performance of our algorithm. Another main aspect comprehends the adaptation of the algorithm to the 3D case, specially in the particular case of Substructure Search in Protein Molecules problem. Finally, we will also improve the implementation of some of our subroutines in order to optimize their processor and memory requirements.

References

[1] H. Alt, K. Mehlhorn, H. Wagener and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete & Computational Geometry*, 3:237–256, 1988.

[2] S. Aluru and F.E. Sevilgen. Dynamic compressed hyperoctrees with application to the N-body problem. *Proc. 19th Conf. Found. Softw. Tech. Theoret. Comput. Sci.*, LNCS 1738:21-33, 1999.

[3] R.P. Brent . Algorithms for Minimization Without Derivatives. *Englewood Cliffs*, NJ: Prentice-Hall, 1973.

[4] V. Choi, N. Goyal. A Combinatorial Shape Matching Algorithm for Rigid Protein Docking. *CPM 2004, LNCS 3109*, pp. 285-296, 2004.

[5] A. Efrat, A. Itai and M.J. Katz. Geometry helps in Bottleneck Matching and related problems. *Algorithmica*, 31:1–28, 2001.

[6] D. Eppstein, M.T. Goodrich, and J.Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. *21st ACM Symp. on Comp. Geom.*, 296–305, 2005.

[7] D. Halperin and M. H. Overmars. Spheres, molecules, and hidden surface removal. *Comput. Geom. Theory Appl.*, 11: 83-102, 1998.

[8] P.J. Heffernan S. Schirra. Approximate decision algorithms for point set congruence. *Computational Geometry: Theory and Applications* 4(3), 137–156, 1994.

[9] K. H. Hunt. Kinematic Geometry of Mechanisms (Chapters 4,7). *Oxford University Press*, 1978.

[10] P. Indyk, S. Venkatasubramanian. Approximate congruence in nearly linear time. *Comput. Geom.* 24(2): 115–128, 2003.

Appendix

The search algorithm

The main search function, denoted search_1, seeks for candidate zones formed by only one node and invokes itself and the other two search functions, called search_2 and search_4 respectively. Consequently, search_2 finds zones formed by pairs of nodes and also launches itself and search_4. Finally, search_4 locates zones formed by quartets of nodes and only invokes itself.

Algorithm 1 Search_1(node N, candidateList List, geometricInformation Info)

```
for all S sons of N do
  if Compatible(S.information, Info) then
    if S.size ≥ Info.size then
      Search_1(S,List,Info)
    else {We have found a candidate node}
      addToCandidateZones(List,candidateZone(S))
    end if
  end if
end for
{Continue in pairs of nodes if necessary (four possibilities)}
for all S_1, S_2 pairs of neighboring sons of N do
  if Compatible( (S_1, S_2).information, Info) then
    if (S_1, S_2).size ≥ Info.size then
      Search_2(S_1, S_2,List,Info)
    else {We have found a candidate pair}
      addToCandidateZones(List,candidateZone(S_1, S_2))
    end if
  end if
end for
{Finally, continue in the quartet formed by the four sons if necessary}
(S_1, S_2, S_3, S_4) ← sonsOf(N)
if Compatible( (S_1, S_2, S_3, S_4).information, Info) then
  if (S_1, S_2, S_3, S_4).size ≥ Info.size then
    Search_4(S_1, S_2, S_3, S_4,List,Info)
  else {We have found a candidate quartet}
    addToCandidateZones(List,candidateZone(S_1, S_2, S_3, S_4))
  end if
end if
```

The algorithm presented here illustrate at high level how function Search_1 works. The parameters correspond respectively to the node of $\mathcal{Q}_{\mathcal{B}}$ being considered (the algorithm starts with a call to Search_1 with the root of $\mathcal{Q}_{\mathcal{B}}$ as first parameter), the list that maintains the candidate zones already detected and finally the geometric information we are looking for (corresponding to the one we have calculated for $\mathcal{A}$).

Some details are omitted from the algorithm for clarity's sake:

- Function "Compatible" comprehends the implementations of all the compatibility criteria for all the parameters being used.
- Function "sonsOf" yields the siblings of a given node.
- All the functions that have a candidate zone between its parameters accept all their three possible subtypes.

Functions Search_2 and Search_4 work similarly, although they present some differences as can be seen in the outlines of their algorithms.

Algorithm 2 Search_2(node N_1, node N_2, candidateList List, geometricInformation Info)

Input: res?

Output: List contains all the possible candidate areas inside the part of $\mathbb{R}^2$ limited by the pair (N_1, N_2).

```
{It is unnecessary to test single nodes}
{Pairs of neighboring nodes with the same father are also already covered}
{Test for pairs of neighboring nodes with different father (two possibilities)}
for all S1, S2 pairs of neighboring sons with different father do
  if Compatible( (S1, S2).information, Info) then
    if (S1, S2).size ≥ Info.size then
      Search_2(S1, S2,List,Info)
    else {We have found a candidate pair}
      addToCandidateZones(List,candidateZone(S1, S2))
    end if
  end if
end for
{quartets of nodes with the same father are already covered}
{Continue in the quartet formed by pairs of sons with different father if necessary}
(S1, S2, S3, S4) ← neighboringSonsOf(N1, N2)
if Compatible( (S1, S2, S3, S4).information, Info) then
  if (S1, S2, S3, S4).size ≥ Info.size then
    Search_4(S1, S2, S3, S4,List,Info)
  else {We have found a candidate quartet}
    addToCandidateZones(List,candidateZone(S1, S2, S3, S4))
  end if
end if
```

Algorithm 3 Search_4(node N_1, node N_2,node N_3, node N_4, candidateList List, geometricInformation Info)

Input: res?

Output: List contains all the possible candidate areas inside the part of $\mathbb{R}^2$ limited by the quartet (N_1, N_2, N_3, N_4).

```
{It is unnecessary to test single nodes, or any kind of pairs}
{quartets of nodes with the same father or with two neighboring fathers are also already covered}
{Continue in the quartet formed by one son of each of (N1, N2, N3, N4)}
(S1, S2, S3, S4) ← neighboringSonsOf(N1, N2, N3, N4)
if Compatible( (S1, S2, S3, S4).information, Info) then
  if (S1, S2, S3, S4).size ≥ Info.size then
    Search_4(S1, S2, S3, S4,List,Info)
  else {We have found a candidate quartet}
    addToCandidateZones(List,candidateZone(S1, S2, S3, S4))
  end if
end if
```