All-Pairs Shortest Paths in Geometric Intersection Graphs

Timothy M. Chan[1] and Dimitrios Skrepetos[2]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, tmc@illinois.edu

[2] Cheriton School of Computer Science, University of Waterloo, dskrepet@uwaterloo.ca

Abstract. We address the All-Pairs Shortest Paths (APSP) problem for a number of unweighted, undirected geometric intersection graphs. We present a general reduction of the problem to static, offline intersection searching (specifically detection). As a consequence, we can solve APSP for intersection graphs of n arbitrary disks in $O\left(n^2 \log n\right)$ time, axis-aligned line segments in $O\left(n^2 \log\log n\right)$ time, arbitrary line segments in $O\left(n^{7/3} \log^{1/3} n\right)$ time, d-dimensional axis-aligned boxes in $O\left(n^2 \log^{d-1.5} n\right)$ time for $d \geq 2$, and d-dimensional axis-aligned unit hypercubes in $O\left(n^2 \log\log n\right)$ time for $d = 3$ and $O\left(n^2 \log^{d-3} n\right)$ time for $d \geq 4$.
In addition, we show how to solve the Single-Source Shortest Paths (SSSP) problem in unweighted intersection graphs of axis-aligned line segments in $O\left(n \log n\right)$ time, by a reduction to dynamic orthogonal point location.

Keywords: shortest paths, geometric intersection graphs, intersection searching data structures, disk graphs

1 Introduction

As a motivating example, consider the following toy problem: given a set S of n axis-aligned line segments in the plane representing a road network, and two points p_1 and p_2 lying on two segments of S, compute a path from p_1 to p_2 that stays on S while minimizing the number of turns. (See Figure 1.)

To solve the problem, we can create a vertex for each segment of S and an (unweighted, undirected) edge between two vertices if their corresponding segments intersect. This defines the *intersection graph* $G(S)$. Then given two points p_1 and p_2, lying on the segments s and t of S, a minimum-turn path from p_1 to p_2 corresponds precisely to an unweighted shortest path from s to t in $G(S)$. Naively constructing $G(S)$ and running breadth-first search (BFS) would require $O(n^2)$ worst-case time. In Section 4, however, we observe an $O(n \log n)$-time algorithm, which is new to the best of the authors's knowledge. In fact, the algorithm solves the more general, Single-Source Shortest Paths (SSSP) problem in $G(S)$, by an application of data structures for *dynamic orthogonal point location* [23,7].

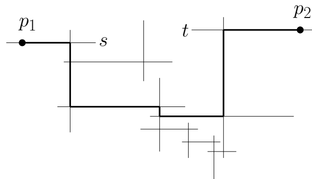

Fig. 1. A set S of axis-aligned line segments is shown. The path from p_1, lying on segment s, to p_2, lying on segment t, staying on S and using the minimum number of turns is marked in bold.

Our main focus in this paper will be a similar problem, namely the All-Pairs Shortest Path (APSP) problem in geometric intersection graphs. More generally, given a set S of n geometric objects, its intersection graph $G(S)$ is defined by creating one vertex for every object of S and an (undirected, unweighted) edge between two vertices if their corresponding objects intersect. We want to compute a representation of an unweighted shortest path between s and t for every pair of objects $s, t \in S$. For general unweighted, undirected graphs the problem can be solved in $O(n^{\omega})$ time (e.g., see [5,28]), where $\omega < 2.373$ is the matrix multiplication exponent [30], but better results are possible for geometric intersection graphs.

Our main results are as follows:

- For arbitrary disks, we solve APSP in $O\left(n^2 \log n\right)$ time. The disk case is naturally motivated by applications in ad hoc communication networks. Following work by Cabello and Jejčič [9] on SSSP for unit-disk graphs, a previous paper by the authors [16] studied APSP for unit-disk graphs and gave an $O\left(n^2\sqrt{\frac{\log\log n}{\log n}}\right)$-time algorithm, but the approach cannot be extended to arbitrary disks. A paper by Kaplan et al. [25] contains an algorithm for SSSP for disks (which can be used for APSP), but this is for a directed variant of intersection graphs (called "transmission graphs"), and the running time has multiple logarithmic factors unless we assume that the maximum-to-minimum radius ratio is bounded.
- For axis-aligned line segments, we solve APSP in $O\left(n^2 \log\log n\right)$ time, which is better than running n times the $O(n \log n)$-time algorithm for SSSP that we have mentioned for the toy problem at the beginning. No previous results have been reported, to the best of the authors's knowledge.
- When the line segments are not axis-aligned but have arbitrary orientations instead, we solve APSP in $O\left(n^{7/3} \log^{1/3} n\right)$ time, which is a little better than the general $O(n^{\omega})$ result, at least with the current upper bound on ω. (Regardless, our algorithm has the advantage of being combinatorial.)
- See Table 1 for further results on axis-aligned boxes, unit hypercubes, and fat triangles of roughly equal size.

All these results stem from one single, general technique, which reduces APSP to the design of data structures for static, offline intersection *detection*, i.e., given a query object, decide whether there is an input object intersecting it (and report

Geometric Objects	Running Time
arbitrary disks	$O(n^2 \log n)$
axis-aligned line segments	$O\left(n^2 \log\log n\right)$
arbitrary line segments	$O\left(n^{7/3} \log^{1/3} n\right)$
d-dimensional axis-aligned boxes	$O\left(n^2 \log^{d-1.5} n\right)$ for $d \geq 2$
d-dimensional axis-aligned unit hypercubes	$O\left(n^2 \log\log n\right)$ for $d = 3$ and $O\left(n^2 \log^{d-3} n\right)$ for $d \geq 4$
fat triangles of roughly equal size	$O\left(n^2 \log^4 n\right)$

Table 1. The results for APSP

one if the answer is yes). Our technique, described in Section 2, works by visiting vertices in an order prescribed by a spanning tree; given the BFS tree from a source vertex s as a guide, we can generate the BFS tree from an adjacent source vertex s' quickly, by exploiting the fact that distances to s' are approximately known up to ± 1, and by using the right geometric data structures. Some form of this simple idea has appeared before for general graphs (e.g., see [4,11]), but it is somehow overlooked by previous researchers in the context of geometric APSP.

To appreciate the advantages of the new technique, we should compare it with other known general approaches:

- First, a naive approach is to solve SSSP n times from every source independently, i.e., generate the BFS trees from each source from scratch. Geometric SSSP problems can often be reduced to *dynamic* data structuring problems, for example, as observed in Chan and Efrat's paper [13] (the reduction is much simplified in the unweighted, undirected setting). In fact, our solution to the toy problem at the beginning is done via this approach. However, dynamic data structures for geometric intersection or range searching usually are more complicated and have slower query times than their static counterparts, sometimes by multiple logarithmic factors. For example, the arbitrary disk case requires dynamic data structures for additively weighted nearest neighbor search, and a BFS therein takes nearly $O\left(n \log^{10} n\right)$ time [26]. Our reduction to *static* data structuring problems yields better results.
- Another general approach is to employ *biclique covers* [21,2] to sparsify the intersection graph first and then solve the problem on the sparsified graph. Biclique covers are related to static, offline intersection searching data structures (e.g., as noted in [10]). However, the complexity of biclique covers also tends to generate extra logarithmic factors. For example, for d-dimensional boxes, the sparsified graph has $O\left(n \log^d n\right)$ edges, leading to an $O\left(n^2 \log^d n\right)$-time algorithm, but our solution requires $O\left(n^2 \log^{d-1.5} n\right)$ time. For arbitrary disks, the complexity of the biclique covers is even worse ($O\left(n^{3/2+\varepsilon}\right)$ [3]), leading to an $O\left(n^{5/2+\varepsilon}\right)$-time algorithm, which is much

slower than our $O\left(n^2 \log n\right)$ result. The underlying issue is that intersection searching (as implicitly needed in biclique covers) may in general be harder than intersection detection.

In deriving our result for axis-aligned boxes, we also obtain a new $O\left(n\sqrt{\log n}\right)$-time algorithm for offline rectangle stabbing in two dimensions (preprocess n axis-aligned rectangles so that we can find a rectangle stabbing each query point). This result (see the full paperAppendix A) may be of independent interest.

For the rest of the paper, for $s, t \in S$, where S is a set of geometric objects, let $dist[s,t]$ denote the distance of the shortest path from s to t in the intersection graph of S and $pred[s,t]$ denote the predecessor of t in that path. In SSSP we want to compute $dist[s,t]$ and $pred[s,t]$ for a given $s \in S$ and $\forall t \in S$, while in APSP we want to compute $dist[s,t]$ and $pred[s,t]$ $\forall s, t \in S$. All algorithms assume the standard unit-cost RAM model of computation where the word size is at least $\log n$ in bits.

2 Reducing APSP to static, offline intersection detection

In this section, we reduce the problem of solving APSP in unweighted, undirected geometric intersection graphs of objects of constant-description complexity to static, offline intersection detection. We assume that the graph is connected; if not, then we can simply work with every connected component independently. We first compute an arbitrary spanning tree T_0 of $G(S)$, root it at an arbitrary object $s_0 \in S$, and then compute the shortest path tree of s_0. Then, we visit each object s of T_0 in a pre-order manner, and compute the shortest path tree of s by using the shortest path tree of s' as a guide, where s' is the parent of s in T_0. The pseudocode of the algorithm is given in Algorithm 1. The initial call is APSP(S, s_0).

Algorithm 1: APSP(S, s_0)

```
1 build G(S)
2 compute any spanning tree T_0 of G(s) and root it at any s_0 ∈ S
3 compute the shortest path tree of s_0
4 for each s ∈ S − {s_0} following a pre-order traversal of T_0 do
5     compute the shortest path tree T(s) of s, using the shortest path tree
        T(s') of its parent s' in T_0, by calling SSSP(S, s, T(s'))
```

It remains to describe how to compute the shortest path tree of a vertex $s \in S$, given the shortest path tree of a vertex t at unit distance from it, i.e., how to implement Line 5 in Algorithm 1. From the triangle inequality and from $dist[s, s'] = 1$, we know that if $dist[s', z] = \ell$ for an object $z \in S$, then $\ell - 1 \leq dist[s,z] \leq \ell + 1$. Thus we already have an 1-additive approximation of the

distances $dist[s', z]$ for any $z \in S$. To compute the exact distances from s to any object $z \in S$, we follow the procedure of the next paragraph.

As in classical BFS, we proceed in $n-1$ steps, where in step ℓ we assume that we have found all the objects at distance at most $\ell - 1$ from s and want to produce the objects at distance exactly ℓ. The objects at distance exactly $\ell - 1$ from s are called the *frontier* objects, while the ones whose distance has not yet been found are called the *undiscovered* objects. Then we need to procure quickly all the undiscovered objects that intersect the frontier objects. Because of the 1-additive approximation, an object z can be at distance ℓ from s only if it is at distance $\ell - 1$, ℓ, or $\ell + 1$ from s'. These points are called the *candidate* objects. Hence we need to determine, for each candidate object, whether it intersects any frontier object. This is an instance of intersection searching, or more specifically, *intersection detection*:

> Preprocess a set of input objects into a data structure so that we can quickly decide if a given query object intersects any input object, and report one such input object if it exists.

In our application, the input objects are *static*, and the query objects are *offline*, i.e., are all given in advance.

To summarize, the pseudocode is presented in Algorithm 2. Thus we obtain the following theorems:

Algorithm 2: SSSP$(S, s, T(s'))$

1 $dist[s, s] = 0$
2 $dist[s, z] = \infty \; \forall z \in S - \{s\}$
3 $pred[s, z] = NULL \; \forall z \in S$
4 **for** $\ell = 0$ *to* $n - 1$ **do**
5 $\quad A_\ell = \{z \mid dist[s', z] = \ell\}$ `// objects at distance` ℓ `from` s'
6 **for** $\ell = 1$ *to* $n - 1$ **do**
7 $\quad F = \{z \in S \mid dist[s, z] = \ell - 1\}$ `// frontier objects`
8 $\quad C = A_{\ell-1} \cup A_\ell \cup A_{\ell+1}$ `// candidate objects`
9 $\quad$ build a static, offline intersection detection data structure for F and C
10 $\quad$ **for** $z \in C$ **do**
11 $\quad\quad$ **if** $dist[s, z] = \infty$ **then**
12 $\quad\quad\quad$ query the data structure for z
13 $\quad\quad\quad$ let w be the answer
14 $\quad\quad\quad$ **if** w *not NULL* **then**
15 $\quad\quad\quad\quad dist[s, z] = \ell$
16 $\quad\quad\quad\quad pred[s, z] = w$

Theorem 1. *Given a set S of n objects of constant-description complexity and the shortest path tree of an object $s' \in S$ in the unweighted, undirected intersection graph of S, we can compute the shortest path tree of an object $s \in S$, where*

dist$[s, s'] = 1$*, in the same graph in* $O(SI(n,n))$ *time, where* $SI(n,m)$ *is the time to construct a static, offline intersection detection data structure for* n *objects and query it* m *times, assuming the property that* $SI(n_1, m_1) + SI(n_2, m_2) \leq SI(n_1 + n_2, m_1 + m_2)$*.*

Proof. Let n_ℓ (resp. m_ℓ) be the number of frontier (resp. candidate) objects in step ℓ of the BFS. During the algorithm, an object is in the frontier exactly once and in the candidate at most thrice; in other words, $\sum_{\ell=1}^{n-1} n_\ell \leq n$ and $\sum_{\ell=1}^{n-1} m_\ell \leq 3n$. Then the time to compute the shortest path tree of s is $O\left(\sum_{\ell=1}^{n-1} SI(n_\ell, m_\ell)\right) = O(SI(n,n))$. □

Theorem 2. *We can solve APSP in an unweighted geometric intersection graph of* n *objects of constant-description complexity in* $O\left(n^2 + nSI(n,n)\right)$ *time, where* $SI(\cdot,\cdot)$ *is defined as in Theorem 1.*

Proof. In Lines 1–3 of Algorithm 1, we can build $G(S)$ in $O\left(n^2\right)$ time, find a spanning tree T_0, and compute the shortest-path tree of s_0, in $O\left(n^2\right)$ time naively. In each of the $n-1$ iterations, Line 3 of Algorithm 1 takes $O\left(SI(n,n)\right)$ time by Theorem 1. □

3 Applications

In this section we apply Theorem 2 and known data structures for static, offline intersection detection to obtain efficient APSP algorithms in specific families of geometric intersection graphs. Some of the data structures we employ are in fact online.

Arbitrary disks in the plane. We first consider intersection graphs of disks of arbitrary radii, also known as disk graphs. The static intersection detection data structure for disks will be based on an additively weighted Voronoi diagram, where the distance between a site w corresponding to a disk of radius r_w and a point x is defined as $d(w,x) = ||w - x|| - r_w$. This Voronoi diagram allows us to determine the disk whose boundary is closest to a query point. We construct the Voronoi diagram for the centers of the frontier disks and a point location data structure for the diagram's cells. Then we query the Voronoi diagram with the center of each query disk. We can check if the query disk and the disk returned by the query intersect in constant time.

The time for building the additively weighted Voronoi diagram of n disks is $O(n \log n)$ [22]. We build a point location data structure in $O(n \log n)$ time, so that (online) queries take $O(\log n)$ time [29]. Therefore, $SI(n,n) = O(n \log n)$.

Theorem 3. *We can solve APSP in an unweighted intersection graph of* n *disks in* $O\left(n^2 \log n\right)$ *time.*

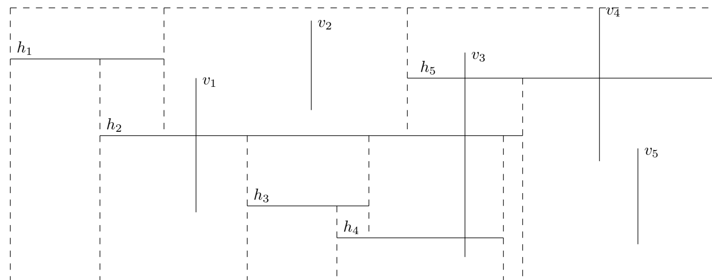

Fig. 2. This figure depicts a set of horizontal input segments, its vertical decomposition (shown by the dashed lines), and a set of vertical query segments.

Axis-aligned line segments in the plane. We now turn our attention to intersection graphs of axis-aligned line segments. We describe a static intersection detection data structure for horizontal input segments and vertical query segments. (Vertical input segments and horizontal query segments can be handled by a symmetric structure.) The data structure is composed of the vertical decomposition of the horizontal input segments, stored in a point location data structure. Given a vertical query segment, we perform a point location query for its bottom endpoint. If the top endpoint lies in the same cell, there is no intersection; otherwise, we can report the segment bounding the top side of the cell. (See Figure 2 for an example.)

We can apply the static orthogonal point location data structure of Chan [12] (Theorem 2.1), with $O(n \log\log U)$ preprocessing time and $O(\log\log U)$ query time, under the assumption that all coordinates are integers bounded by U. Thus, $SI(n, n) = O(n \log\log U)$. This implies an APSP algorithm running in $O\left(n^2 \log\log U\right)$ time. At the beginning, we can presort all coordinates in $O(n \log n)$ time and replace each coordinate value with its rank; this ensures that $U = n$. Thus, we obtain:

Theorem 4. *We can solve APSP in an unweighted intersection graph of n axis-aligned line segments in $O\left(n^2 \log\log n\right)$ time.*

The result can be easily be extended to any set of line segments with a constant number of different orientations.

Arbitrary line segments. Next we consider the case of arbitrary line segments. Chazelle [17] (Theorem 4.4) has given an $O\left(n^{4/3} \log^{1/3} n\right)$-time algorithm to count the number of intersections among n line segments. The algorithm can be modified to count the number of intersections between n red (input) line segments and n blue (offline query) line segments. In fact, it is straightforward to adapt the algorithm to decide, for each blue segment, whether it intersects any red segment and, if yes, report one such red segment. Thus, $SI(n, n) = O\left(n^{4/3} \log^{1/3} n\right)$.

Theorem 5. *We can solve APSP in an unweighted intersection graph of n arbitrary line segments in $O\left(n^{7/3}\log^{1/3} n\right)$ time.*

Axis-aligned boxes in d dimensions. For the case of axis-aligned rectangles in $d = 2$ dimensions, offline rectangle intersection counting is known to be reducible [19] to offline orthogonal range counting, for which Chan and Pătraşcu [15] have given an $O\left(n\sqrt{\log n}\right)$-time algorithm, under the assumption that all coordinates have been presorted. Consequently, we can decide for each query box whether it intersects any input box. With more effort, we can adapt their technique to report a witness input box for each query box with a yes answer, and thus solve offline intersection detection in $SI(n, n) = O\left(n\sqrt{\log n}\right)$ time; see Appendix A for details. At the beginning, we can presort all coordinates in $O(n \log n)$ time.

For axis-aligned boxes in $d \geq 3$ dimensions, we can use standard range trees [18] with the above $d = 2$ base case to obtain $SI(n, n) = O\left(n \log^{d-1.5} n\right)$.

Theorem 6. *We can solve APSP in an unweighted intersection graph of n d-dimensional axis-aligned boxes in $O\left(n^2 \log^{d-1.5} n\right)$ time for $d \geq 3$.*

Axis-aligned unit hypercubes in d dimensions. When the axis-aligned boxes are unit hypercubes, the time bound for offline intersection detection can be improved. We build a uniform grid with unit side length and solve the problem inside each grid cell separately. Each input or query unit hypercube participates in at most a constant (2^d) number of grid cells. Inside a grid cell, each unit hypercube is effectively unbounded along d sides. Without loss of generality, we may assume that each input box is of the form $(-\infty, a_1] \times \cdots \times (-\infty, a_d]$ and each query box is of the form $[b_1, \infty) \times \cdots \times [b_d, \infty)$. Thus, the problem reduces to offline *dominance* detection: decide for each query point $(b_1, \ldots, b_d)$ whether it is dominated by some input point $(a_1, \ldots, a_d)$ and, if yes, report one such input point.

For $d = 3$, Gupta et al. [24] gave an algorithm to answer n offline dominance reporting queries in $O((n + K) \log \log U)$ time where K is the total output size, under the assumption that all coordinates are integers bounded by U. Their algorithm can be easily adapted to answer n offline dominance detection queries in $O(n \log \log U)$ time. This implies an APSP algorithm running in $O\left(n^2 \log \log U\right)$ time. At the beginning, we can presort all coordinates in $O(n \log n)$ time and replace each coordinate value with its rank; this ensures that $U = n$.

For $d \geq 4$, Afshani et al. [1] (following Chan et al. [14]) gave a deterministic algorithm to answer n offline dominance reporting queries in $O\left(n \log^{d-3} n + K\right)$ time where K is the total output size. It can be checked that their algorithm can answer n offline dominance detection queries in $O\left(n \log^{d-3} n\right)$ time. (One step in their algorithm which involves reversing the role between input and query points becomes unnecessary for the detection problem.)

Theorem 7. *We can solve APSP in an unweighted intersection graph of n d-dimensional axis-aligned unit hypercubes in $O\left(n^2 \log\log n\right)$ time for $d = 3$ and in $O\left(n^2 \log^{d-3} n\right)$ time for $d \geq 4$.*

Fat triangles in the plane. Finally we consider the intersection graph of *fat* triangles (i.e., triangles that have bounded inradius-to-circumradius ratios) with roughly equal size. Katz [27] (Theorem 4.1 (i) and (iii)) has given an (online) data structure achieving $SI(n,n) = O\left(n \log^4 n\right)$. Thus:

Theorem 8. *We can solve APSP in an unweighted intersection graph of n fat triangles with roughly equal size in $O\left(n^2 \log^4 n\right)$ time.*

4 Reducing SSSP to decremental intersection detection

We give in this section a reduction of SSSP in intersection graphs to dynamic intersection detection.

We will emulate the classic BFS algorithm in the following way. Let $s \in S$ be the given source vertex. We proceed iteratively in $n-1$ steps and follow the same process in each one. In step ℓ we assume that we have found all the distances and predecessors for all the objects that are at distance no more than $\ell - 1$ from s. We employ the definitions of the frontier and undiscovered objects as given in Section 2. The goal is to compute the distances and predecessors for all the undiscovered objects that are at distance ℓ from s. Those objects are the ones that have at least one intersection with a frontier object. To find those intersections we maintain an intersection detection data structure for the undiscovered objects that supports deletion—a deletion-only dynamic data structure is often referred to as a *decremental* data structure. We query the structure with the frontier objects; each time we detect an intersection of a frontier object with an undiscovered one, we properly update the latter's distance and predecessor and delete it from the data structure. The pseudocode of the algorithm is given in Algorithm 3.

We conclude this section with the following theorem.

Theorem 9. *We can solve SSSP in an unweighted, undirected geometric intersection graph in $O(DI(n,n))$ time, where $DI(n,m)$ is the time to construct a decremental intersection detection data structure of n objects and perform n deletions and m queries.*

Proof. The correctness of the algorithm can be easily proved by induction.

For the running time, we notice that an object can be in the frontier in only one step of the algorithm. In the beginning all the objects except the source are undiscovered (thus in the decremental intersection detection data structure as well), and once an object is deleted from that set, it is never inserted again. When querying the intersection detection data structure with a frontier object t there are two possible outcomes. If the query returns an undiscovered object z that intersects t, then z is deleted from the data structure, and since it is

Algorithm 3: SSSP(S, s)

```
1  dist[s,s] = 0
2  dist[s,t] = ∞ ∀ t ∈ S − {s}
3  pred[s,t] = NULL ∀ t ∈ S
4  build a decremental intersection detection data structure for S − {s}
     // i.e., for undiscovered objects
5  for ℓ = 1 to n − 1 do
6      F = {t ∈ S | dist[s,t] = ℓ − 1}                       // frontier
7      for each t ∈ F do
8          while true do
9              query the data structure with t
10             let z be the answer
11             if z not NULL then
12                 dist[s,z] = ℓ
13                 pred[s,z] = t
14                 delete z from the data structure
15             else
16                 break
```

never reinserted, this type of query happens only once $\forall z \in S$. If the query returns nothing, then this is the last query that t performs in that step, and since t can be in the frontier at most once, this type of query happens only once $\forall t \in S$. Consequently the total number of queries in the data structure is $O(n)$. Furthermore, the number of deletions in the decremental data structure is obviously $O(n)$. Thus the total running time is $O(DI(n, n))$. □

Application to axis-aligned line segments. We need a decremental intersection detection data structure for horizontal input segments and vertical query segments. (Vertical input segments and horizontal query segments can be handled by a symmetric structure.) Giyora and Kaplan [23] (Theorem 5.3) and Blelloch [7] (Theorem 6.1) provided a data structure for supporting vertical ray shooting queries in $O(\log n)$ time and insertions and deletions of horizontal segments in $O(\log n)$ time—the problem is sometimes referred to as *dynamic orthogonal point location*. This immediately implies $DI(n, n) = O(n \log n)$. Thus:

Theorem 10. *We can solve SSSP in an unweighted intersection graph of n axis-aligned line segments in $O(n \log n)$ time.*

The result can be easily be extended to any set of line segments with a constant number of different orientations.

5 Conclusion

Interesting open problems in unweighted, undirected geometric intersection graphs include constructing efficient distance oracles and computing the diameter in truly subquadratic $O\left(n^{2-\varepsilon}\right)$ time for some $\varepsilon > 0$, in view of Cabello's

recent breakthrough for the diameter problem in planar graphs [8]. For certain geometric objects such as arbitrary line segments, even a quadratic-time APSP algorithm is already open. Finally, solving APSP in the weighted case seems to be more difficult, as we can no longer exploit the general reduction from APSP to static, offline intersection detection.

References

1. Peyman Afshani, Timothy M. Chan, and Konstantinos Tsakalidis. Deterministic rectangle enclosure and offline dominance reporting on the RAM. In *Proceedings of the Forty-First Annual International Colloquium on Automata, Languages, and Programming*, pages 77–88, 2014.
2. Pankaj K. Agarwal, Noga Alon, Boris Aronov, and Subhash Suri. Can visibility graphs be represented compactly? *Discrete & Computational Geometry*, 12(3):347–365, 1994.
3. Pankaj K. Agarwal, Marco Pellegrini, and Micha Sharir. Counting circular arc intersections. *SIAM Journal on Computing*, 22(4):778–793, 1993.
4. Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.
5. Noga Alon, Zvi Galil, Oded Margalit, and Moni Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proceedings of the Thirty-Third Annual IEEE Symposium on Foundations of Computer Science*, pages 417–426, 1992.
6. Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana A. Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 572–591, 2015.
7. Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 894–903, 2008.
8. Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2143–2152. SIAM, 2017.
9. Sergio Cabello and Miha Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry*, 48(4):360–367, 2015.
10. Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM Journal on Computing*, 36(3):681–694, 2006.
11. Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Transactions on Algorithms*, 8:1–17, 2012.
12. Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Transactions on Algorithms*, 9(3):22, 2013.
13. Timothy M. Chan and Alon Efrat. Fly cheaply: On the minimum fuel consumption problem. *Journal of Algorithms*, 41(2):330–337, 2001.
14. Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Computational Geometry*, pages 1–10, 2011.
15. Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the Twenty-First Annual ACM-SIAM Aymposium on Discrete Algorithms*, pages 161–173, 2010.

16. Timothy M. Chan and Dimitrios Skrepetos. All-pairs shortest paths in unit-disk graphs in slightly subquadratic time. In *Proccedings of the Twenty-Seventh Annual International Symposium on Algorithms and Computation*, pages 24:1–24:13, 2016.
17. Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9(2):145–158, 1993.
18. Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4/5):177–181, 1981.
19. Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14(3):124–127, 1982.
20. David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 827–835, 2001.
21. Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.
22. Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
23. Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3):28, 2009.
24. Prosenjit Gupta, Ravi Janardan, Michiel H. M. Smid, and Bhaskar DasGupta. The rectangle enclosure and point-dominance problems revisited. *International Journal of Computational Geometry and Applications*, 7(5):437–455, 1997.
25. Haim Kaplan, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Spanners and reachability oracles for directed transmission graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 34, 2015.
26. Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2495–2504, 2017.
27. Matthew J. Katz. 3-D vertical ray shooting and 2-D point enclosure, range searching, and arc shooting amidst convex fat objects. *Computational Geometry*, 8(6):299–316, 1997.
28. Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.
29. Jack Snoeyink. Point location. In *Handbook of Discrete and Computational Geometry*, pages 767–785. CRC Press, second edition, 2004.
30. Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, pages 887–898, 2012.

A Appendix: Offline Rectangle Intersection Detection

Rectangle intersection detection (finding some input rectangle intersecting a query rectangle) can be easily reduced [18] to

(i) axis-aligned line segment intersection detection (finding some input horizontal/vertical segment intersecting a query vertical/horizontal segment),
(ii) orthogonal range detection (finding some input point inside a query rectangle), and
(iii) *rectangle stabbing* detection (finding some input rectangle containing a query point).

In Section 4, we have already noted how to answer n offline queries of type (i) in $O(n \log \log U)$ time, assuming that coordinates are integers bounded by U. Babenko et al. [6] have already described how to adapt Chan and Pătraşcu's technique [15] to answer n offline queries of type (ii) in $O\left(n\sqrt{\log n}\right)$ time (Babenko et al. actually solved the *range successor* problem, which is equivalent to finding the lowest point in a 3-sided query rectangle unbounded from above—it is easy to see that 4-sided orthogonal range detection reduces to this problem). We now describe how to adapt Chan and Pătraşcu's technique to answer n offline queries of type (iii) in $O\left(n\sqrt{\log n}\right)$ time. This result is of independent interest. (Chan et al. [14] noted a similar result but only for the case of *disjoint* rectangles.)

Theorem 11. *Given n input axis-aligned rectangles and query points in the plane whose coordinates have been pre-sorted, we can report, for each query point q, an input rectangle containing q (if exists) in $O\left(n\sqrt{\log n}\right)$ time.*

Proof. Let w be the word size. Without loss of generality, assume that all y-coordinates are distinct.

Special case: all x-coordinates are small integers bounded by s. We use a divide-and-conquer resembling a binary interval tree in x, and incorporate bit-packing techniques.

The input to our recursive algorithm is a list of the vertices of the input rectangles and query points, arranged in bottom-to-top order. We do not explicitly store the y-coordinates, just the x-coordinates of the points in this list. The number of words in the input list is thus $O((n \log s)/w)$. The output is represented as a list storing the minimum and maximum x-coordinates of one rectangle containing each query point, in bottom-to-top order of the query points; some of the entries may be null. The number of words in the output list is $O((n \log s)/w)$.

The algorithm proceeds as follows. Let $x = m$ be the vertical line that divides the x-universe into two halves of length $s/2$. Let R_m be the subset of rectangles intersecting $x = m$. We first compute the union of R_m (a y-monotone polygon, or multiple such polygons). This subproblem reduces to computing the left/right envelope of a set of vertical line segments; Eppstein and Muthukrishnan [20] have solved this subproblem in linear time $O(|R_m|)$, assuming that coordinates have been pre-sorted in x and y. In our case, the coordinates have already been sorted in y, and we can sort in x by counting sort in $O(|R_m| + s)$ time. We can then solve the problem for R_m and Q by a bottom-to-top scan, using $O((n \log s)/w)$ additional word operations. Next, let R_ℓ (resp. R_r) be the subset of rectangles completely to the left (resp. right) of $x = m$, and

let Q_ℓ (resp. Q_r) be the subset of query points to the left (resp. right) of $y = m$. We recursively solve the subproblem for R_ℓ and Q_ℓ and the subproblem for R_r and Q_r. The input to either subproblem can be formed by a linear scan using $O((n \log s)/w)$ word operations, and the output can be merged by another linear scan using $O((n \log s)/w)$ word operations.

Excluding the cost of computing the unions of the R_m's, the total running time is $O((n \log^2 s)/w + s \log s)$, since there are $O(\log s)$ levels of recursion. Observe that each rectangle lies in exactly one subset R_m over the entire recursion tree. Thus, the total cost of computing the unions of the R_m's is $O(n)$.

One remaining issue is that the output only records the x-coordinates of the reported rectangles. To retrieve the y-coordinates, we first partition the original set of input rectangles into $O(s^2)$ classes with a common minimum and maximum x-coordinates. For each query point, we have identified one class which contains an answer. For each class χ, we can gather its input rectangles R_χ and query points Q_χ, both pre-sorted by y, and answer these queries; this is a 1-dimensional problem in y (finding an input interval containing each query point), which is a special case of the above-mentioned envelope problem and can be solved in linear time $O(|R_\chi| + |Q_\chi|)$. The total extra time over all classes χ is thus $O(n)$.

We conclude that the special case case can be solved in $O(n + (n \log^2 s)/w)$ time, assuming that $n \geq s^2$.

General case. We use a divide-and-conquer resembling a degree-s segment tree. We use $s - 1$ vertical lines to divide the plane into s slabs each with $O(n/s)$ rectangle vertices and query points. Each rectangle can be divided into at most three parts, where the left (resp. right) part is contained in one of the s slabs, and the middle part has x-coordinates aligned with the dividing vertical lines. For the middle parts, we can round the x-coordinates of the query points to align with the dividing lines and apply the algorithm for the above special case in $O(n + (n \log^2 s)/w)$ time. For the left and right parts, we recursively solve the subproblems inside the s slabs. The answers can be combined in $O(n)$ time.

Each rectangle and each query point participates in $O(\log_s n)$ recursive calls. The total time is thus $O((n + (n \log^2 s)/w) \cdot (\log n / \log s))$. Setting $\log s = \sqrt{w}$ yields $O(n(\log n)/\sqrt{w})$, assuming that $n \geq 2^{\Omega(\sqrt{w})}$. One final issue is that the algorithm has used exotic word operations on w-bit words. If we set $w = \delta \log n$ for a sufficiently small constant δ, we can replace these operations by table lookup after an initial precomputation in $2^{O(w)} = n^{O(\delta)}$ time. □