ALL-PAIRS SHORTEST PATHS IN GEOMETRIC INTERSECTION GRAPHS*

Timothy M. Chan† *and Dimitrios Skrepetos*‡

Abstract. We present a simple and general algorithm for the all-pairs shortest paths (APSP) problem in unweighted geometric intersection graphs. Specifically we reduce the problem to the design of static data structures for offline intersection detection. Consequently we can solve APSP in unweighted 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 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 $d \geq 4$, and d-dimensional axis-aligned boxes in $O\left(n^2 \log^{d-1.5} n\right)$ time for $d \geq 2$.

We also reduce the single-source shortest paths (SSSP) problem in unweighted geometric intersection graphs to decremental intersection detection. Thus, we obtain an $O\left(n \log n\right)$-time SSSP algorithm in unweighted intersection graphs of n axis-aligned line segments.

1 Introduction

As a motivating example, consider the following toy problem. Given a set S of n axis-aligned line segments that represents a road network and two points p_1 and p_2 that lie on two segments of S, compute a path on S from p_1 to p_2 such that the number of turns is minimized. See Figure 1.

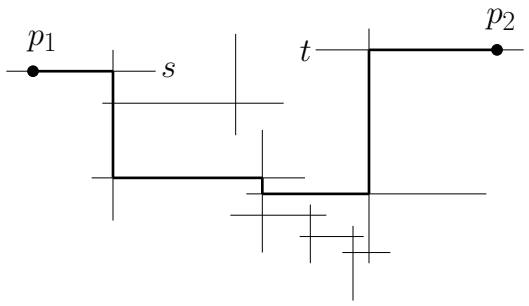


Figure 1: An instance of the toy problem.

To solve this toy problem, we could build the *intersection graph* G of S, i.e., create a vertex for each segment of S and an unweighted, undirected edge between two vertices if their

*The conference version of this paper appeared in WADS 2017 [16].

†*Department of Computer Science, University of Illinois at Urbana-Champaign*, tmc@illinois.edu

‡*Cheriton School of Computer Science, University of Waterloo*, dskrepet@uwaterloo.ca

corresponding segments intersect. Then given two points p_1 and p_2 that lie on the segments s and t of S, a p_1-to-p_2 minimum-turn path corresponds to an s-to-t shortest path in G. We could find such a path with breadth-first search (BFS) in G in $O\left(n^2\right)$ time. However, we observe in Section 4 an $O\left(n \log n\right)$-time algorithm by applying data structures for *dynamic orthogonal point location* [25, 7], which has not been noticed before. Our algorithm in fact solves the more general single-source shortest paths (SSSP) problem: given a source vertex, compute the shortest paths from it to all vertices.

Our main focus in this paper is a problem similar to SSSP, namely the all-pairs shortest path (APSP) problem, where we want to compute all pairwise shortest paths. We study APSP in unweighted geometric intersection graphs, defined similarly to the above paragraph. In general unweighted, undirected graphs the problem can be solved in $O\left(n^\omega\right)$ time (e.g., see [5, 31]), where $\omega < 2.373$ is the matrix multiplication exponent [32], but better results are possible for geometric intersection graphs.

Our main results are as follows.

- In arbitrary disk graphs, naturally motivated by applications in ad-hoc communication networks, we can solve APSP in $O\left(n^2 \log n\right)$ time. Following work by Cabello and Jejčič [8] on SSSP in unit-disk graphs, a previous paper by the authors [15] gave an $O\left(n^2 \sqrt{\frac{\log \log n}{\log n}}\right)$-time algorithm therein, but the approach cannot be extended to arbitrary disk graphs. For arbitrary disk graphs, the dynamic Voronoi diagrams data structure of Kaplan et al. [28] can be employed to solve the problem in nearly $O\left(n^2 \log^{12} n\right)$ time,[1] as explained in the next page.
- In axis-aligned line segments, we can solve APSP in $O\left(n^2 \log \log n\right)$ time, which is better than running n times our $O\left(n \log n\right)$-time SSSP algorithm that we mentioned earlier. No previous results for the problem 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 can solve APSP in $O\left(n^{7/3} \log^{1/3} n\right)$ time. Our result is a little better than the $O\left(n^\omega\right)$-time solutions [5, 31] for general unweighted, undirected graphs, at least with the current upper bound on $\omega < 2.373$ [32, 30]. Regardless, our algorithm has the advantage of being combinatorial.
- See Table 1 for further results in intersection graphs of axis-aligned boxes, unit hypercubes, and fat triangles of roughly equal size.

All these results stem from a simple and general technique, described in Section 3. Specifically we reduce APSP to static, offline intersection *detection*: given a query object, decide whether there is an input object that intersects it (and report one if the answer is yes). Our algorithm visits the vertices of the given graph in an order prescribed by a spanning

[1] Another paper by Kaplan et al. [27] also described similar results with multiple extra logarithmic factors for a *directed* variant of disk intersection graphs, called "transmission graphs", assuming that the maximum-to-minimum disk radii ratio is bounded.

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

tree T and generates the shortest-path tree of each vertex s' by using the shortest-path tree from its parent s in T as a guide. To do that quickly, we exploit the fact that distances from s' are approximately known up to ± 1 and also employ the right geometric data structures. Some form of this simple idea has appeared before for general graphs (e.g., see [4, 10]), but it is somehow overlooked by previous researchers in the context of geometric APSP.

Our solution compares favorably with the two previous general methods for the problem.

- The first general method runs an SSSP algorithm from every source independently by a reduction to *dynamic* data structuring problems, as observed by the ideas of Chan and Efrat [12]. Actually the reduction is much simplified in the unweighted, undirected setting. 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 employs dynamic data structures for additively weighted nearest-neighbor search. Thus solving the problem that way takes nearly $O\left(n^2 \log^{12} n\right)$ time [28], while our approach requires only $O\left(n^2 \log n\right)$ time.
- The second general method employs *biclique covers* [23, 2] to sparsify the intersection graph and then applies an SSSP algorithm from each vertex. However, biclique covers are related to static, offline intersection searching (e.g., as noted in [9]), which is generally harder than intersection detection. For example, the sparsified intersection graph of d-dimensional boxes 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 $O\left(n^2 \log n\right)$ solution.

To derive our result for intersection graphs of axis-aligned boxes, we describe in Section 5 a new $O\left(n\sqrt{\log n}\right)$-time algorithm for offline rectangle stabbing in two dimensions: preprocess n axis-aligned rectangles and n points, such that we can find a rectangle (if any) that stabs each query point. That data structure may be of independent interest.

2 Preliminaries

2.1 Definitions and notations

Recall that an unweighted geometric intersection graph G is the intersection graph of a set S of geometric objects. That is, vertices correspond to objects and edges to pairwise intersections. We assume that these objects are of constant-description complexity, that G is represented implicitly by S (thus it can be stored with only linear space), and that G, without loss of generality, is connected.

Let S be a set of geometric objects, and let G be the unweighted geometric intersection graph it defines. For any $s, t \in S$ we let $\pi[s,t]$ denote an s-to-t shortest path in G and let $dist[s,t]$ denote its length. We also refer to $dist[s,t]$ as shortest-path distance or simply distance. By $pred[s,t]$ we denote t's predecessor on $\pi[s,t]$. Also, we define the shortest-path tree $T(s)$ of $s \in S$ to be a spanning tree of G rooted at s, such that for each $t \in S$ the s-to-t shortest-path distance in $T(s)$ corresponds to $dist[s,t]$.

2.2 Model of computation

Throughout the paper, we use the standard (real) RAM model of computation. Specifically, we have random access to an array of words, each storing a real number, a $\Theta(\log n)$-bit integer (where n is the input size), or a pointer to another word. Moreover, we can perform any standard arithmetic operation, such as addition, subtraction, multiplication, division, and comparison, that involves a constant number of words in constant time. We assume that we can compute square roots of real numbers exactly in constant time.

We can support custom operations in $(\delta \log n)$-bit words in $O(1)$ time, where $0 < \delta < 1$ is a constant, after $o(n)$ preprocessing time. To do so, we can perform the desired operation to each of the $2^{\delta \log n} = n^{\delta} = o(n)$ possible inputs and store the results in a lookup table (assuming that each operation can naively be performed in $O\left(\log^{O(1)} n\right)$ time).

3 Reducing APSP to static, offline intersection detection

Here, we reduce APSP in unweighted, undirected geometric intersection graphs to static, offline intersection detection. We first build an arbitrary spanning tree T_0 of G, root it at an arbitrary object $s_0 \in S$, and compute the shortest-path tree of s_0. Then, we visit each object s of T_0 in a pre-order manner and compute its shortest-path tree by using that of s' as a guide, where s' is the parent of s in T_0. See Algorithm 1 for the pseudocode.

It remains to describe how to compute the shortest-path tree from an object $s \in S$, given the shortest-path tree from its parent s' in T_0 (Line 5 in Algorithm 1). We first notice that from the triangle inequality and from $dist[s,s'] = 1$, if $dist[s',z] = \ell$ for some $z \in S$, then $\ell - 1 \leq dist[s,z] \leq \ell + 1$. That is, we already have an ± 1-additive approximation of the distances from s; see Figure 2. To compute the exact distances from s, we devise a BFS-like algorithm of $n-1$ steps, where in the beginning of each step ℓ, we assume that we have properly processed each $t \in S$ at distance at most $\ell - 1$ from s. We call the rest

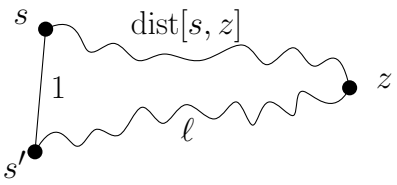
Algorithm 1: GeoAPSP(S)

```
1 build the unweighted, undirected geometric intersection graph G of S
2 compute any spanning tree T_0 of G and root it at any s_0 ∈ S
3 compute T(s_0)
4 for each s ∈ S − {s_0} following a pre-order traversal of T_0 do
5     compute T(s) by using T(s') as a guide, where s' is the parent of s in T_0
6 return T(·)
```

of the objects *undiscovered* and those at distance exactly $\ell - 1$ from s *frontier*. However, contrary to the classical BFS, in each step ℓ we do not have to consider all undiscovered objects. Specifically, the ± 1-additive approximation of the distances from s implies that the only undiscovered objects that can be at distance ℓ from it are the ones at distance $\ell - 1$, ℓ, or $\ell + 1$ from s'. We call these *candidate* objects. Thus, we need to find for each candidate whether there is any frontier object that intersects it. This is an instance of the following problem:

Subproblem 1. (Intersection Detection) *Preprocess a set of input objects into a data structure, such that we can quickly decide if a given query object intersects any input object, and, if it does, report any such object.*

The input (frontier objects) is *static*, and the queries (candidate objects) are *offline*, i.e., are all given in advance. The pseudocode is presented in Algorithm 2.



$$\ell - 1 \leq \mathrm{dist}[s, z] \leq \ell + 1$$

Figure 2: ± 1-additive approximation.

Theorem 1. (Computing shortest-path trees with ± 1-additive approximation) *Given a set S of n objects and the shortest-path tree of $s' \in S$ in the unweighted, undirected intersection graph of S, we can compute the shortest-path tree from an neighbor $s \in S$ of s' in $O(SI(n, n))$ time, where $SI(n, m)$ is the time to construct a static, offline intersection detection data structure (Subproblem 1) for n objects and query it m times. We assume that $SI(n_1, m_1) + SI(n_2, m_2) \leq SI(n_1 + n_2, m_1 + m_2)$.*

Proof. As discussed above, the ± 1 approximation of the distances from s implies that the candidate objects are the only undiscovered objects that need to be considered in each step. Thus, the correctness of our algorithm follows from that of classical BFS.

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

```
1  dist[s, s] = 0
2  dist[s, z] = ∞ ∀ z ∈ S − {s}
3  pred[s, z] = NULL ∀ z ∈ S
4  for ℓ = 0 to n − 1 do
5      A_ℓ = {z | dist[s', z] = ℓ}
6  for ℓ = 1 to n − 1 do
7      F = {z ∈ S | dist[s, z] = ℓ − 1}
8      C = A_{ℓ−1} ∪ A_ℓ ∪ A_{ℓ+1}
9      build a static, offline intersection detection data structure with input set
         F (frontier objects) and query set C (candidate objects)
10     for t ∈ C do
11         if dist[s, t] = ∞ then
12             query the data structure for t
13             let w be the answer
14             if w not NULL then
15                 dist[s, t] = ℓ
16                 pred[s, t] = w
17 return T(s)
```

Let n_ℓ (respectively m_ℓ) be the number of frontier (respectively candidate) objects in the step ℓ of our algorithm. An object is a frontier object exactly once and a candidate object at most thrice, i.e., $\sum_{\ell=1}^{n-1} n_\ell \leq n$ and $\sum_{\ell=1}^{n-1} m_\ell \leq 3n$. Thus, the time to compute the shortest-path tree from s is $O\left(\sum_{\ell=1}^{n-1} SI(n_\ell, m_\ell)\right) = O\left(SI(n, n)\right)$. □

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

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

3.1 Applications

In this section, we use known results to provide static, offline intersection detection data structures for a family of geometric objects and then apply Theorem 2 to solve APSP in the corresponding intersection graphs. Let $SI(\cdot, \cdot)$ be as in Theorem 1.

Arbitrary disks. Given a set of n disks, we create an additively weighted Voronoi diagram and a point location data structure for the cells of that diagram. The sites are the disk centers, and the weight of each site is equal to the radius of the corresponding disk, i.e.,

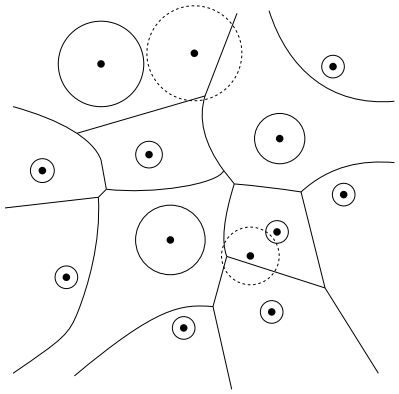


Figure 3: A set of input disks (solid), their additively weighted Voronoi diagram (not an accurate one), and two query disks (dashed).

the distance between a site p corresponding to a disk of radius r_p and a point q is defined as $d(p,q) = ||p-q|| - r_p$. To detect an intersection of a query disk with an input disk of radius r_q that is centered at q, we can find the input point p that minimizes $d(p,q)$ and check whether $d(p,q) \leq r_q$. Employing the additively-weighted Voronoi diagram algorithm of Fortune [24] together with the point location data structure of Edelsbrunner et al. [19], we have $SI(n,n) = O(n \log n)$. See Figure 3.

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

Axis-aligned line segments. To construct a static, offline intersection data structure for n axis-aligned line segments we can without loss of generality consider only horizontal input segments and only vertical query segments. We build the vertical decomposition of the former and store it in a point location data structure. Then, given a vertical query segment, we perform a point location query with its endpoints. If they lie in the same cell, there is no intersection. Otherwise, we pick the cell that contains the bottom endpoint of the query segment and report the input segment that bounds its upper side. See Figure 4.

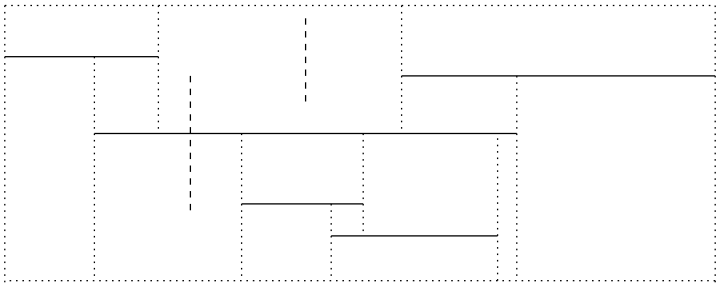


Figure 4: A set of horizontal input segments (solid), its vertical decomposition (dotted), and two query vertical segments (dashed).

The point location data structure we employ is the static orthogonal point location data structure of Chan [11, Theorem 2.1], requiring $O(n \log \log U)$ preprocessing and

$O(\log\log U)$ query time for n input segments, where U is the universe size. By presorting all coordinates in $O(n\log n)$ time and replacing each coordinate with its rank, we ensure that $U = n$. Hence, $SI(n,n) = O(n\log\log n)$.

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 by constructing one instance of Chan's data structure per orientation.

Arbitrary line segments. The $O\left(n^{4/3}\log^{1/3} n\right)$-time algorithm of Chazelle [18, Theorem 4.4] for counting the number of intersections among n line segments can be extended to count the number of intersections between n red (input) line segments and n blue (offline query) line segments. In fact, the algorithm can decide whether each blue segment intersects any red segment, and if so, 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. Offline rectangle intersection counting in n axis-aligned rectangles in the plane is known to be reducible [21] to offline orthogonal range counting, for which Chan and Pătraşcu [14, Corollary 2.3] have given an $O\left(n\sqrt{\log n}\right)$-time algorithm, assuming that the x- and y-coordinates of all the vertices of the rectangles have been presorted. Thus, we can decide for each query box whether it intersects any input box. In Section 5, we adapt the technique of Chan and Pătraşcu to construct a data structure that can also report such an input box if it exists. Hence, after presorting the coordinates in $O(n\log n)$ time, we have that $SI(n,n) = O\left(n\sqrt{\log n}\right)$. For axis-aligned boxes in $d \geq 3$ dimensions, we use standard range trees [20] with the above planar 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. We can construct more efficient static, offline data structures when the n boxes are unit hypercubes. Specifically, we first build a uniform grid with unit side length and then 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, and inside each of them, it is effectively unbounded along d sides. We assume without loss of generality that each input box is of the form $(-\infty, a_1] \times \cdots \times (-\infty, a_d]$ and that 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 $d = 3$, the algorithm of Gupta et al. [26, Theorem 3.1] answers n offline dominance reporting queries in $O((n + K) \log \log U)$ time, where n is the input size, and K is the total output size, assuming that all coordinates are integers bounded by U. Thus, n offline dominance detection queries can be answered in $O(n \log \log U)$ time. By presorting all coordinates in $O(n \log n)$ time and replacing each with its rank, we ensure that $U = n$. Hence, $SI(n, n) = O(n \log \log n)$. For $d \geq 4$, Afshani et al. [1, Remark in Section 5] (following Chan et al. [13]) has given a deterministic algorithm to answer n offline dominance reporting queries in $O\left(n \log^{d-3} n + K\right)$ time, where n is the input size, and K is the total output size. It can be checked that n offline dominance detection queries can be answered in $O\left(n \log^{d-3} n\right)$ time.

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. Given n *fat* triangles (i.e., triangles of bounded inradius-to-circumradius ratios) of roughly equal sizes, Katz [29, Theorem 4.1(i)] has shown how to construct an intersection reporting data structure of $O\left(n \log^4 n\right)$ preprocessing and $O\left(\log^3 n + K \log^2 n\right)$ query time, where n is the input size, and K is the total output size (here $K \leq 1$). Hence, $SI(n, n) = O\left(n \log^4 n\right)$.

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

4 Reducing SSSP to decremental intersection detection

We now show how to reduce SSSP in unweighted, undirected geometric intersection graphs to decremental intersection detection. Our approach is again to use a BFS-like algorithm. See Algorithm 3 for the pseudocode. As in classical BFS, we want to find every undiscovered object that shares an edge with a frontier object, but inspecting all edges incident to the latter ones would require quadratic total time. However, in geometric intersection graphs, an edge between two objects exists if and only if they intersect each other, so it suffices to find for each undiscovered object one, if any, frontier object that intersects it. To do so, we would like to solve the *decremental* version of Subproblem 1, i.e., the input undergoes only deletions.

An object is undiscovered only until we find a frontier object that intersects it. Thus, we can employ a decremental intersection detection data structure for the undiscovered objects and query it in each step of the algorithm with the frontier objects. Whenever we detect an intersection, we properly set the distance and predecessor of the relevant undiscovered object and delete it from the data structure.

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

Algorithm 3: GeoSSSP(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}
5  F = {s}
6  for ℓ = 1 to n − 1 do
7      F' = ∅
8      for each z ∈ F do
9          while true do
10             query the data structure with z and let t be the answer
11             if t not NULL then
12                 dist[s, t] = ℓ
13                 pred[s, t] = z
14                 delete t from the data structure
15                 F' = F' ∪ {t}
16             else
17                 break
18     F = F'
19 return dist[s, ·] and pred[s, ·]
```

Proof. In unweighted, undirected geometric intersection graphs, there is an edge between two objects if and only if they intersect, so the correctness of our algorithm follows from that of BFS. The running time is dominated by the time required to construct, update, and query the decremental intersection detection data structure of the undiscovered objects. Initially, each object except the source is undiscovered, and once we delete it from the data structure, we never reinsert it. Thus, the total number of deletions is $O(n)$. There are two cases for a query with a frontier object z. If an undiscovered object t that intersects z is returned, we delete t from the data structure and never reinsert it, so this case happens once for each $t \in S$. If nothing is returned, z does not perform any other queries in that step and, since z is a frontier point only once, this case also happens once $\forall z \in S$. Thus, the total number of queries is $O(n)$, implying that our algorithm takes $O\left(DI(n, n)\right)$ time. □

Application to axis-aligned line segments. In intersection graphs of axis-aligned line segments, we need a decremental intersection data structure for horizontal input segments and vertical query segments. Vertical input segments and horizontal query segments can be handled with a symmetrical structure. We can use the data structure of either Giyora and Kaplan [25, Theorem 5.3] or of Blelloch [7, Theorem 6.1], both of which support vertical ray shooting queries and insertions and deletions of horizontal segments in logarithmic time. Thus $DI(n, n) = O\left(n \log n\right)$.

Theorem 10. (SSSP in unweighted intersection graphs of axis-aligned line segments) *We can solve SSSP in an unweighted intersection graph of n axis-aligned line segments in*

$O(n \log n)$ *time.*

The above theorem can easily be extended to any set of line segments with a constant number of different orientations by constructing one instance of the data structure of Giyora and Kaplan or of Blelloch per orientation.

Remark: Recently, Chan and Tsakalidis [17] improved the results of Giyora and Kaplan and of Blelloch to support vertical ray shooting queries in $O\left(\frac{\log n}{\log \log n}\right)$ time and updates in near $O\left(\sqrt{\log n}\right)$ time, assuming that the coordinates are polynomially-bounded integers. Consequently, the time bound in Theorem 10 can be improved to $O\left(n \cdot \frac{\log n}{\log \log n}\right)$.

5 Static, offline rectangle intersection detection

We now show how to construct a data structure for the static, offline rectangle intersection detection problem: given a set of n input and query rectangles, find for each query rectangle one, if any, input rectangle that intersects it. We assumed the existence of such a structure to obtain the APSP algorithm of Theorem 6 for unweighted, undirected graphs of d-dimensional boxes. As depicted in Figure 5, we can easily reduce that problem [20] to the following subproblems:

(i) Axis-aligned line segment intersection detection: finding some input horizontal (respectively vertical) segment that intersects a query vertical (respectively horizontal) segment.

(ii) Orthogonal range detection: finding some input point inside a query rectangle.

(iii) *Rectangle stabbing* detection: finding some input rectangle that contains a query point.

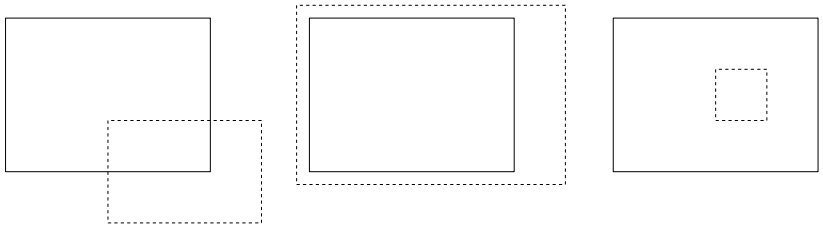


Figure 5: Axis-aligned line segment intersection detection, orthogonal range detection, and rectangle stabbing detection. The input rectangles are solid, while the query rectangles are dashed.

In Section 3.1, we showed how to answer n offline queries of type (i) in $O(n \log \log U)$ time, assuming that the coordinates of the vertices of the rectangles are integers bounded by U. Also, Babenko et al. [6] have already described how to adapt Chan and Pătraşcu's technique [14] 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. Chan et al. [13] noted a similar result but only for the case of pairwise *disjoint* rectangles.

Theorem 11. (Rectangle Stabbing Detection) *Given n input axis-aligned rectangles and n planar query points with presorted coordinates, we can report for each query point, an input rectangle, if any, that contains it in $O\left(n\sqrt{\log n}\right)$ time.*

Proof. We use words of $w = \delta \log n$ bits, where $0 < \delta < 1$ is a constant. Also, we assume, without loss of generality, that all y-coordinates are pairwise distinct.

Special case: all x-coordinates are small integers bounded by s. We employ a divide-and-conquer scheme that resembles a binary interval tree in the x-axis and bit-packing techniques. The input of our algorithm is the list of the x-coordinates of the vertices of the input rectangles and of the query points, presorted by y. Thus, it can be packed with $O\left((n \log s)/w\right)$ words. The output is represented as the list of the minimum and maximum x-coordinates of a rectangle, if any, that contains each query point in bottom-to-top order. Hence, it can also be packed with $O\left((n \log s)/w\right)$ words.

Let R be the set of input rectangles and Q that of query points. First, we find the vertical line $x = m$ that divides the x-universe into two halves of length $s/2$. Let R_m be the subset of rectangles that intersect that line. The union of R_m is a y-monotone polygon or multiple such polygons, and we can find it by computing the left/right envelope of the vertical line segments of the input rectangles. Eppstein and Muthukrishnan [22] have shown how to do that in $O\left(|R_m|\right)$ time, assuming that the coordinates have been pre-sorted in x and y. In our case, the latter has already been done, and to obtain the former we employ counting sort in $O\left(|R_m| + s\right)$ time. Then, we solve the problem for R_m and Q with a bottom-to-top scan, using $O\left((n \log s)/w\right)$ additional word operations.

Next, let R_ℓ (respectively R_r) be the subset of rectangles completely to the left (respectively right) of $x = m$ and Q_ℓ (respectively Q_r) the subset of query points to the left (respectively 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 $O\left((n \log s)/w\right)$ word operations, and the output can be merged by another linear scan using $O\left((n \log s)/w\right)$ word operations.

Excluding the cost of computing the unions of the R_m's, the total running time is $O\left((n \log^2 s)/w + s \log s\right)$ since there are $O\left(\log s\right)$ levels of recursion. Each rectangle lies in exactly one subset R_m over the entire recursion tree, so 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 $O\left(s^2\right)$ classes with 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 $O(|R_\chi| + |Q_\chi|)$ time, thus the total time over all classes χ is linear. We conclude that the special case case can be solved in $O\left(n + (n \log^2 s)/w\right)$ time, assuming that $n \geq s^2$.

General case. We again use a divide-and-conquer approach, but this time in a degree-s-segment-tree manner. 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 (respectively right) part is contained in one of the s slabs, and the middle part has x-coordinates aligned with the dividing vertical lines. For all 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, if $n \geq s^2$, we recursively solve the subproblems inside the s slabs. We can combine the answers in linear time.

Each rectangle and each query point participates in $O(\log_s n)$ recursive calls. The total time is thus $O((n + (n \log^2 s)/w) \cdot (\log n / \log s))$, which becomes $O(n(\log n)/\sqrt{w})$ by setting $\log s = \sqrt{w}$ and assuming that $n \geq 2^{\Omega(\sqrt{w})}$. Since $w = \delta \log n$, where $0 < \delta < 1$ is a constant, we can perform each of the above word operations in constant time with table lookup, after an initial precomputation in $o(n)$ time (see Section 2.2). We can also handle the base case, $n = s^2$, in linear total time, again with table lookup. □

Theorem 12. (Static, offline rectangle intersection detection) *Given n input and n query axis-aligned rectangles with presorted coordinates, we can report an input rectangle for each query rectangle (if it exists) that intersects it in $O\left(n\sqrt{\log n}\right)$ time.*

6 Conclusion

The obvious open problems are to improve the logarithmic factors in our near-quadratic APSP algorithms, and to obtain a near-quadratic-time APSP algorithm for intersection graphs of arbitrary line segments.

A problem similar to APSP is computing the diameter of a graph, defined as the longest shortest-path distance of any two vertices. It is worth investigating whether we can compute the diameter faster than solving APSP for some of the unweighted geometric intersection graphs considered in this paper.

References

[1] Peyman Afshani, Timothy M. Chan, and Konstantinos Tsakalidis. Deterministic rectangle enclosure and offline dominance reporting on the RAM. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, 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 33rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 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 26th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 572–591, 2015.

[7] Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 894–903, 2008.

[8] Sergio Cabello and Miha Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry*, 48(4):360–367, 2015.

[9] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[10] Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Transactions on Algorithms*, 8:1–17, 2012.

[11] Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Transactions on Algorithms*, 9(3):22, 2013.

[12] Timothy M. Chan and Alon Efrat. Fly cheaply: On the minimum fuel consumption problem. *Journal of Algorithms*, 41(2):330–337, 2001.

[13] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 26th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[14] Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the 26th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 161–173, 2010.

[15] Timothy M. Chan and Dimitrios Skrepetos. All-pairs shortest paths in unit-disk graphs in slightly subquadratic time. In *Proccedings of the 27th International Symposium on Algorithms and Computation (ISAAC)*, pages 24:1–24:13, 2016.

[16] Timothy M. Chan and Dimitrios Skrepetos. All-pairs shortest paths in geometric intersection graphs. In *Proceeding of the 15th International Symposium on Algorithms and Data Structures (WADS)*, pages 253–264, 2017.

[17] Timothy M. Chan and Konstantinos Tsakalidis. Dynamic planar orthogonal point location in sublogarithmic time. In *Proceedings of the 34th Symposium on Computational Geometry (SoCG)*, pages 25:1–25:15, 2018.

[18] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9(2):145–158, 1993.

[19] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[20] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4-5):177–181, 1981.

[21] Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14(3):124–127, 1982.

[22] David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 827–835, 2001.

[23] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[24] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[25] Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3):28, 2009.

[26] 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.

[27] Haim Kaplan, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Spanners and reachability oracles for directed transmission graphs. In *Proceedings of the 31st Symposium on Computational Geometry (SoCG)*, pages 156–170, 2015.

[28] 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 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2495–2504, 2017.

[29] 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.

[30] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

[31] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[32] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith–Winograd. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC)*, pages 887–898, 2012.