Reactive Proximity Data Structures for Graphs

David Eppstein, Michael T. Goodrich, and Nil Mamano(✉)

Department of Computer Science, University of California, Irvine, USA
{eppstein,goodrich,nmamano}@uci.edu

Abstract. We consider data structures for graphs where we maintain a subset of the nodes called *sites*, and allow proximity queries, such as asking for the closest site to a query node, and update operations that *enable* or *disable* nodes as sites. We refer to a data structure that can efficiently react to such updates as *reactive*. We present novel reactive proximity data structures for graphs of polynomial expansion, i.e., the class of graphs with small separators, such as planar graphs and road networks. Our data structures can be used directly in several logistical problems and geographic information systems dealing with real-time data, such as emergency dispatching. We experimentally compare our data structure to Dijkstra's algorithm in a system emulating random queries in a real road network.

1 Introduction

Proximity data structures are well-known in computational geometry [10], where sites are points in the plane and distance is measured, e.g., by the Euclidean metric. In this paper, we are interested in proximity data structures for graphs, where sites are defined by a distinguished subset of the vertices and distance is measured by shortest-path distance in the graph. That is, we assume a graph, G, is given and fixed (like a road network for a geographic region) and that distance is measured by shortest paths in this graph. With respect to updates, a non-distinguished vertex in G can be *enabled* to become a site or an existing site can be *disabled* to no longer be a site, and we want our data structure to *react* to such updates so as to be able to quickly answer proximity queries, such as nearest-neighbor or closest-pair queries for sites.

Definition 1 (Reactive proximity). *Given a set, U, known as the* universe, *and a distance function, $d(*)$, for elements in U, maintain a subset, $P \subseteq U$, of* sites, *allowing the following operations:*

- ***Proximity Queries. nearest-neighbor:*** *given a query element $q \in U$, return a site p in P minimizing $d(q, p)$;* ***closest-pair:*** *return a pair, $p, q \in P$, minimizing $d(p, q)$;* ***bichromatic-closest-pair:*** *suppose we have $P = R \cup B$, where $R \cap B = \emptyset$. Then, return a pair, $p, q \in P$, minimizing $d(p, q)$, such that $p \in R$ and $q \in B$.*

© Springer International Publishing AG, part of Springer Nature 2018
M. A. Bender et al. (Eds.): LATIN 2018, LNCS 10807, pp. 777–789, 2018.
https://doi.org/10.1007/978-3-319-77404-6_56

– ***Updates. enable:*** *add an element from U to P;* ***disable:*** *remove an element from P.*

We refer to data structures that can support such queries and updates as *reactive* data structures,[1] in that we have a fixed universe of objects which can be enabled or disabled, and we need to quickly *react* to such events as updates. In this paper, we study the case where the set U is the set of nodes of a graph, and d is the shortest-path distance.

1.1 Applications

There are a number of interesting applications for reactive proximity data structures for graphs, including several logistical problems in geographic information systems dealing with real-time data. Consider, for instance, an application to connect drivers and clients in a private-car service, such as Uber or Lyft, or even a future driverless car service. The data structure could maintain the set of cars waiting at various locations in a city to be put into service. When a client requires a driver, she queries the data structure to find the car nearest to her. This car is then disabled (i.e., it is no longer available) until it completes the trip for this client, at which point the car is then enabled (i.e., it is available) at this new location. Alternatively, we could consider a similar application in the context of police or emergency dispatching, where the data structure maintains the locations of a set of available first responder vehicles. In Sect. 3, we experiment with this type of system emulating random queries in a real road network.

As another example dealing with geo-spatial data, which the authors explore in a companion paper [22], suppose we are given a set of sites representing the locations of certain facilities, such as post offices or voting locations. We wish to partition the vertices of the graph into geographic regions, one for each facility, such that each region has a specified size (in number of nodes) and the shapes of the regions satisfy certain compactness criteria. As we show in the companion paper, a greedy matching algorithm can exploit an efficient reactive data structure to quickly build such a partitioning of the graph.

Reactive proximity data structures can also be useful in other domains, such as content distribution networks, like the one maintained by Akamai. For instance, the data structure could maintain the set of nodes that contain a certain file of interest, like a movie. When another node in the network needs this information, the data structure could be used to find the closest node that can transfer it. Updates allow us to model how copies of such a file migrate in the network, e.g., for load balancing, so that we enable a node when it gets a copy of the file and disable a node when it passes it to another server.

---

[1] We also call such data structures *reactive* to distinguish the kinds of updates we allow (changes to the subset of distinguished vertices) from *dynamic* data structures in which the structure of the graph itself can change, e.g., by vertex or edge insertions or deletions.

Moreover, having a reactive proximity data structure for graphs could allow us to design interesting algorithms that rely on the existence of such data structures. For instance, we discuss how to use them to solve the geometric stable roommates problem [2] in Sect. 2.

1.2 Our Results

In this paper we present a family of reactive data structures for answering proximity problems in graphs. The data structures we present all use a technique based on graph separator hierarchies and they apply to any graphs for which such hierarchies can be built. A *separator* in a given n-vertex graph is a subset of nodes such that removing it partitions the remaining graph into two disjoint subgraphs, each of size at most $2n/3$. The *size* of a separator is the number of nodes in this subset. A *separator hierarchy* is the result of recursively subdividing a graph by using separators. It has been shown recently that classes of graphs with separators of size $O(n^c)$, for any $c < 1$, coincide with the classes of graphs of polynomial expansion [15]. Thus, any graph in this family is suitable for our data structures.

Graphs of polynomial expansion are sparse, but some sparse graphs (such as bounded degree expanders) do not have polynomial expansion. Nonetheless, many important sparse graph families have polynomial expansion. One of the first classes that was shown to have sublinear separators is the class of planar graphs, which have $O(n^{0.5})$-size separators [36]. Separators of the same asymptotic size have also been proven to exist in k-planar graphs [14], bounded-genus graphs [29], minor-closed graph families [34], graphs with sparse crossing graphs [25], and the graphs of certain four-dimensional polyhedra [20].

While road networks are not quite planar because of bridges and underpasses, experimental results show that they can be modeled by graphs with sparse crossing graphs. These graphs, like planar graphs, have $O(n^{0.5})$-size separators [25]. This is fortunate, because it allows our data structures to be used in geographic information systems, e.g., for real-world road networks, as we explore experimentally in this paper.

Not surprisingly, the main technical challenge faced in designing efficient reactive proximity data structures for graphs lies in their reactive nature. In a variant where the sites (i.e., the nodes in P) are fixed, there is a well known solution: the graph-based Voronoi diagram, which maintains the closest site to each node in the graph [26]. With this information, queries can be answered in constant time. However, the Voronoi diagram is not easy to update, requiring $O(n \log n)$ time in sparse graphs with n nodes, which is the same time as for creating the diagram from scratch. If we optimize for update time instead, we could avoid maintaining any information and answer queries directly using a shortest-path algorithm from the query node. Updates would take constant time; queries could be answered using Dijkstra's algorithm [9], which runs in $O(n \log n)$ time in sparse graphs and in $O(n)$ time for graphs with a known separator hierarchy [32]. However, this is clearly not a good solution either if we want fast query times.

Our solutions amount to finding a "sweet spot" between these two extremes. Our novel data structure for the reactive nearest-neighbor problem supports queries in $O(n^{0.5})$ time and updates in $O(n^{0.5} \log\log n)$ time when the underlying graph is a planar graph or a road network. More generally, if the graph belongs to a family with separators of size $S(n) = O(n^c)$, for some $0 < c < 1$, queries and updates run in $O(S(n))$ and $O(S(n) \log\log n)$ time, respectively. Moreover, since forests have separators of size one, the data structure can be shown to have $O(\log n)$ query time and $O(\log n \log\log n)$ update time when the underlying graph is a forest or, more generally, when it has bounded treewidth.

1.3 Prior Related Work

Knuth's discussion of the classic post office problem has given rise to a long line of research on structures for spatial partitioning for answering nearest-neighbor queries, including an entire literature on the topic of Voronoi diagrams [3,8]. Given a collection of sites, these diagrams partition a space into regions such that the points in each region have a particular site as their nearest. Furthermore, Erwig [26] shows that Voronoi diagrams can be extended to graphs and that such structures can be constructed using Dijkstra's shortest-path algorithm (e.g., see [9,31]). These Voronoi diagram structures are efficient for instances when the set of sites is fixed, but they tend to perform poorly for cases in which sites can be inserted or removed. Such *dynamic nearest neighbor* problems have been addressed in geometric settings. For exact two-dimensional dynamic nearest neighbors, a data structure with $O(n^{\epsilon})$ update and query time was given by Agarwal *et al.* [1], and improved to $O(\log^6 n)$ by Chan [6], and to $O(\log^5 n)$ by Kaplan *et al.* [33]. Because of the high complexities of these methods, researchers have also looked at finding approximate nearest neighbors in dynamic point sets. For example, dynamic versions of quadtrees and k-d trees are known [38], and the skip-quadtree data structure can answer approximate nearest neighbor queries and updates in logarithmic time [23].

The graph-based variant of the dynamic nearest neighbor problem that we study falls into the area of *dynamic graph algorithms*, the subject of extensive study [21]. There has been much research on shortest paths in dynamic graphs, e.g., see [4,5,12,13,35,37]. However, previous work has primarily focused on edge insertion and deletion updates rather than the vertex enable/disable updates that we study. Exceptions are the work of Eppstein on maintaining a dynamic subset of vertices in a sparse graph and keeping track of whether it is a dominating set [19], and the work of Italiano and Frigioni on dynamic connectivity for subsets of vertices in a planar graph [28], but these are very different problems from the proximity problems that we study here. To the best of our knowledge, no one has considered maintaining nearest-neighbor data structures for graphs subject to enabling and disabling of sites.

Separator hierarchies, the main technique used in this paper, have proven useful for solving many graph problems [27,30]. Of these, the most related to our problem is the $O(n)$-time single-source shortest path problem for planar graphs when edge weights are non-negative [32]. The same algorithm applies

more generally to graphs that have $O(n^{0.5})$-size separators and for which the separator hierarchy can be built in $O(n)$ time.

2 Reactive Nearest-Neighbor Data Structure for Graphs

We consider undirected graphs with nonnegative edge weights. We begin by describing the concept of a separator hierarchy.

Recall that a *separator* in a given n-vertex graph is a subset $\mathcal{S}$ of nodes such that the removal of $\mathcal{S}$ (and its incident edges) partitions the remaining graph into two disjoint subgraphs (with no edges from one to the other), each of size at most $2n/3$. It is allowed for these subgraphs to be disconnected; that is, removing $\mathcal{S}$ can partition the remaining graph into more than two connected components, as long as those components can be grouped into two subgraphs that are each of size at most $2n/3$. A *separator hierarchy* is the result of recursively subdividing a graph by using separators. Note that since children have size at most $2/3$ the size of the parent, the separator hierarchy is a binary tree of $O(\log n)$ height.

Small separators are necessary for the efficiency of our data structure. As we mentioned, the analysis will depend on the size of the separators for a particular graph family, which we denote by $S(n)$, and which we assume to be of the form n^c with $0 < c < 1$ (because, e.g., forests have separators of size one, and that changes the analysis).

The reactive nearest-neighbor data structure that we describe in this section yields the following theorem.

Theorem 1. *Given an n-node graph from a graph family with separators of size $S(n) = n^c$, with $0 < c < 1$, which can be constructed in $O(n)$ time, it is possible to initialize a reactive data structure that uses $O(nS(n))$ space, in $O(nS(n))$ time, that answers nearest-neighbor queries in $O(S(n))$ time and updates in $O(S(n) \log n)$ time. Alternatively, the data structure could have $O(nS(n) \log n)$ initialization time, $O(S(n))$ query time, and $O(S(n) \log \log n)$ update time.*

2.1 Preprocessing

Initially, we are given the graph $G = (V, E)$ and the subset $P \subseteq V$ of sites. The creation of our data structure consists of two phases. The first phase does not depend on the choice of the subset P of sites, while the second phase incorporates our knowledge of P into the data structure. Note that there are two kinds of nodes of interest: separator nodes and sites. The two sets may intersect, but should not be confused.

Site-independent phase. First, we build a *separator hierarchy* of the graph. This hierarchy can be constructed in $O(n)$ time and space in planar graphs [30] and road networks [25].
Second, we compute, for each graph in the hierarchy, the distance from each separator node to every other node. This computation can be represented as a collection of single-source shortest-path problems, one for each separator node.

As we already mentioned, each single-source shortest path problem can be solved in $O(n)$ time in graphs with $O(n^{0.5})$-size separators and for which we can build a separator hierarchy in linear time [32]. Therefore, in such graphs the time to compute shortest-path distances at the top level of the hierarchy is $O(nS(n))$. We can analyze the time to compute these distances at all levels of the hierarchy by the recurrence

$$T(n) = T(x) + T(y) + O(nS(n)),$$

where x and y are the sizes of two subgraphs, chosen so that $x + y \leq n$, $\max(x, y) \leq 2n/3$, and (among x and y obeying these constraints) so that $T(x) + T(y)$ is maximum. The recurrence is dominated by its top-level $O(nS(n))$ term, and has a solution that is also $O(nS(n))$.

Site-dependent phase. For each graph H in the separator hierarchy, and each separator node s in H, we initialize a priority queue Q_s. The elements stored in Q_s are the sites that belong to H, and their priorities are their distances from s. If we implement the priority queue as a binary heap, constructing each queue Q_s takes linear time. Thus, the time at the top level of the hierarchy is linear per separator node, and the total time analysis of this phase is $O(nS(n))$ as before.

Adding the space and time for the two phases together gives $O(nS(n))$ for planar graphs, or for other graphs on which we can use the linear-time separator-hierarchy-based shortest path algorithm. If we instead use the simpler Dijkstra's algorithm to compute shortest paths, the running time becomes the slightly slower bound of $O(nS(n) \log n)$, plus the time to build the separator hierarchy.

2.2 Queries

Given a node q, we find two sites: the closest site to q with (a) paths restricted to the same side of the partition as q, and with (b) paths containing at least one separator node. The paths considered in both cases cover all possible paths, so one of the two found sites will be the overall closest site to q.

- To find the site satisfying condition (a), we can relay the query to the subgraph of the separator hierarchy containing q. This satisfies the invariant that the query node is a node of the graph. This case does not arise if q is a separator node.
- To find the site satisfying condition (b), we need the shortest path from q to any site, but only among paths containing separator nodes. Note that if a shortest path goes through a separator s, it should end at the site closest to s. Therefore, the length of the shortest path starting at q, going through s, and ending at any site, is $d(q, s) + d(s, \min(Q_s))$, where $\min(Q_s)$ denotes the element with the smallest key in Q_s. We can find the site satisfying condition (b) by considering all the separator nodes and retaining the one minimizing this sum.

The time to find paths of type (b) is $O(S(n))$, since there are $O(S(n))$ separator nodes to check and each takes constant time, as we precomputed all the needed distances. Therefore, the time to find all paths of types (a) and (b) can be analyzed by the recurrence

$$T(n) \leq T(2n/3) + O(S(n)),$$

where the $T(2n/3)$ bound dominates the actual time for recursing in a single subgraph of the separator hierarchy. The solution to this recurrence is $O(S(n))$, when $S(n)$ is polynomial, and therefore the time per query is $O(S(n))$ in this case. If $S(n)$ is constant or polylogarithmic, then the query time is $O(S(n) \log n)$.

We can also implement a heuristic optimization for queries so that we do not need to check every separator node to find a node satisfying condition (b). At each graph of the separator hierarchy, we can sort, for each node, all the separators by distance. This increases the space used by the data structure by a constant factor. Then, after obtaining the recursive candidate satisfying condition (a), to find the second candidate, we consider the separator nodes in order by distance to the query node q. Suppose p is the closest site we have found so far. As soon as we reach a separator node s such that $d(q, s) \geq d(q, p)$, we can stop and ignore the rest of separator nodes, since any site reached through them would be further from q than p. In our experiments (Sect. 3), this optimization reduced the average query runtime by a factor between 1.5 and 9.5, depending on the number of sites. It is more effective when there are many sites, as then the closest site will tend to be closer than many separators at the upper levels of the hierarchy.

2.3 Updates

Suppose that we wish to enable or disable a node p from the set of sites P. Note that, when we perform such an update, the structures computed during the site-independent preprocessing phase (the separator hierarchy and the computation of distances) do not change, as they do not depend on the choice of P. However, we will need to update Q_s for every separator node s in the top-level separator $\mathcal{S}$, by adding or removing p (according to whether we are adding or removing it from P).

Moreover, if p is not a separator node, it will belong to one of the two recursive subgraphs in the separator hierarchy. In this case, we also need to update the data structure for the subgraph containing p recursively, since p will appear in the priority queues of the separator nodes in that subgraph.

The time to add or remove p in all top-level priority queues is $O(\log n)$ per priority queue, for a total time of $O(S(n) \log n)$ at the top level. Again, if we formulate and solve a recurrence for the running time at all levels of the separator hierarchy, this time will be dominated by the top level time, giving a total time of $O(S(n) \log n)$ per update.

We can also obtain an asymptotically faster update time of $O(S(n) \log \log n)$ by, for each separator vertex s, replacing the distances from s to all other nodes

by the ranks of these distances in the sorted list of distances. That is, if the set of distances in sorted order from s to the other nodes are

$$d_1, d_2, d_3, \ldots$$

with $d_1 < d_2 < d_3 < \cdots$, we could replace these numbers by the numbers

$$1, 2, 3, \ldots$$

without changing the comparison between any two distances. This replacement would allow us to use a faster integer priority queue, such as a van Emde Boas tree [16], in place of the binary heap representation of each priority queue Q. However, in order to use this optimization, we need to add the time to sort the distances in the preprocessing time, which increases to $O(nS(n) \log n)$ (assuming a comparison sort is used).

2.4 Additional Applications

We remark that our dynamic nearest neighbor data structure can be extended to directed graphs. The only required change is to compute distances *from* and *to* every separator node. To obtain the latter, we can use Dijkstra's algorithm in the reverse graph.

In addition, the conga line data structure of Eppstein [18] solves the dynamic closest-pair problem with $O(\log n)$ query time, $O(T(n) \log n)$ insertion time, and $O(T(n) \log^2 n)$ deletion time, where $T(n)$ is the time per operation (query or update) of a dynamic nearest-neighbor data structure. Therefore, by combining the data structure in this paper with the conga line data structure, we obtain a data structure for the reactive closest pair problem in graphs with separators of size $S(n) = O(n^c)$, with $0 < c < 1$, that achieves $O(\log n)$ query time, $O(S(n) \log n \log \log n)$ enable time, and $O(S(n) \log^2 n \log \log n)$ disable time.

Furthermore, by combining the data structure in this paper with the data structure from [17], we obtain the same running times for the reactive bichromatic closest-pair problem as for the reactive closest-pair problem. Not using our data structure would result in linear or super-linear times for either queries or updates.

The nearest-neighbor data structure can also be used for the metric stable roommates problem, extending the work of Arkin *et al.* [2] to graphs. In the original problem, we wish to match a set of points in a geometric space so that there is no unmatched pair, (p, q), such that p and q are both closer to each other than the points they are matched to. For points in general position, they show that a simple greedy algorithm, which repeatedly matches and removes a closest-pair of points, will produce a solution to the geometric stable roommates problem. An efficient algorithm for this problem is the nearest-neighbor chain algorithm, which solves it in $O(|P|)$ queries and updates of a reactive nearest-neighbor data structure [22]. Hence, combined with our data structure, we can solve the greedy matching problem (for metric stable roommates in graphs) in $O(nS(n) + |P|S(n) \log n)$ time. Without our data structure, using a shortest-path algorithm in the nearest-neighbor chain algorithm increases this time to $\Omega(|P|n)$.

Finally, note that forests and graphs with bounded treewidth have separators of size O(1), e.g., see [7]. If we reformulate and solve the recurrence equations accounting for the fact that there is constant number of separator nodes, we obtain an $O(n \log n)$ preprocessing time, $O(\log n)$ query time, and $O(\log^2 n)$ update time (or $O(\log n \log \log n)$ using an integer priority queue as discussed above).

3 Experiments

In this section, we evaluate our data structure empirically on real-world road network data, the Delaware road network from the DIMACS dataset [11]. We consider the biggest connected component of the network, which has 48812 nodes and 60027 edges. This dataset has been planarized: overpasses and underpasses have been replaced by artificial intersection nodes. Each trial in our experiment begins with a number of uniformly distributed random sites, and then performs 1000 operations. We consider the cases of only queries, only updates, and a mixture of both (see Fig. 1). The updates alternate between enables and disables, whereas the operations in the mixed case alternate between queries and updates. We compare the performance of our data structure against a basic data structure that simply uses Dijkstra's algorithm for the queries.

3.1 Implementation Details

We implemented the algorithms in Java 8.[2] We then executed them and timed them as run on an Intel Core CPU i7-3537U 2.00 GHz with 4 GB of RAM, under Windows 10.

We implemented the optimization for queries described in Sect. 2.2, and compared it with the unoptimized version in order to evaluate if its worth the extra space. For updates, we used a normal binary heap, as these tend to perform better in practice than more sophisticated data structures.

A factor that affects the efficiency of the data structure is the size and balance of the separators. Our hierarchy for the Delaware road network had a total of 504639 nodes across 8960 graphs up to 13 levels deep. Among these graphs, the biggest separator had 81 nodes. Rather than implementing a full planar separator algorithm to find the separators (recall that the data had been planarized), we choose the smallest of two simply-determined separators: the vertical and horizontal lines partitioning the nodes into two equal subsets. While these are not guaranteed to have size $O(\sqrt{n})$, past experiments on the transversal complexity in road networks [24] indicate that straight-line traversals of road networks should provide separators with low complexity, making it unnecessary to incorporate the extra complexity of a full planar graph separator algorithm.

When a separator partitions a graph in more than two connected components, we made one child per component. Thus, our hierarchy is not necessarily a binary tree, and may be shallower. We set the base case size to 20. At the base case, we perform Dijkstra's algorithm. Experiments with different base-case sizes did not affect the performance significantly.

[2] The source code is available at github.com/nmamano/NearestNeighborInGraphs.

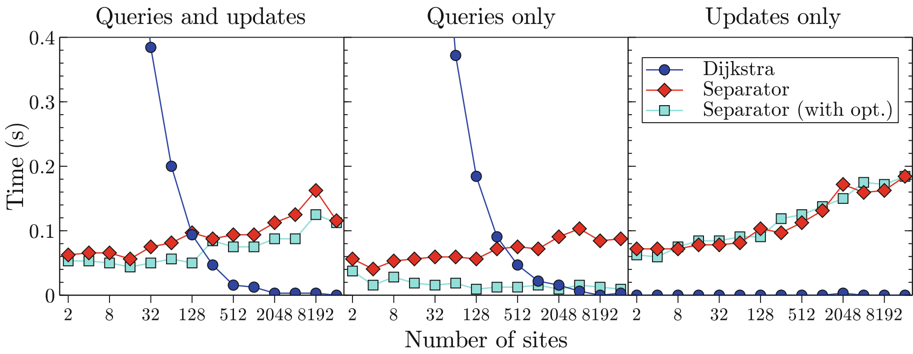

Fig. 1. Time needed by the data structures to complete 1000 operations in the Delaware road network [11] for a range of number of sites (in a logarithmic scale). Each data point is the average of 5 runs with different sets of random sites (the same sets for all the algorithms).

3.2 Results

Figure 1 depicts the results. Table 1 shows the corresponding data for the case of mixed operations, which is the case of interest in a reactive model.

- The runtime of Dijkstra's algorithm is roughly proportional to the number of sites, because with more sites it requires less exploration to find the closest one. Moreover, initialization and updates require virtually no time. Thus, this choice is superior for large numbers of sites, while being orders of magnitude slower when the number of sites is low (see Table 1).
- The data structure based on a separator hierarchy is not affected as much by the number of sites. The update runtime only increases slightly with more sites because of the operations with bigger heaps. This is consistent with its asymptotic runtime, which is $O(S(n)\log n)$ for any number of sites. The optimization, which reduces the number of separators needed to be checked, can be seen to have a significant effect on queries, especially as the number of sites increases: it is up to 9.5 times faster on average with 2048 sites. However, since it has no effect in updates, in the mixed model with the same number of updates and queries the improvement is less significant.
- The data structure requires a significant amount of time to construct the hierarchy. Our code constructed the hierarchy for the Delaware road network in around 15 s. Fortunately, this hierarchy only needs to be built once per road network. The limiting factor is the space requirement of $O(n\sqrt{n})$, which caused us to run out of memory for other road networks from the DIMACS dataset with over 10^5 nodes.

Table 1. Time in milliseconds needed by the data structures to complete 1000 operations (mixed queries and updates) in the Delaware road network for a range of number of sites (in a logarithmic scale). Each data point is the average, minimum, and maximum, of 5 runs with different sets of random sites (the same sets for all the algorithms).

# sites	Dijkstra	Separator	Separator (with opt.)
2	3797 (3672 – 3906)	63 (47 – 94)	53 (31 – 94)
4	2303 (2203 – 2359)	66 (63 – 78)	53 (47 – 63)
8	1272 (1250 – 1297)	66 (47 – 78)	50 (47 – 63)
16	694 (641 – 781)	56 (47 – 63)	44 (31 – 47)
32	384 (359 – 406)	75 (63 – 94)	50 (47 – 63)
64	200 (172 – 219)	81 (63 – 94)	56 (47 – 63)
128	94 (94 – 94)	97 (94 – 109)	50 (47 – 63)
256	47 (47 – 47)	88 (78 – 94)	84 (78 – 109)
512	16 (16 – 16)	94 (94 – 94)	75 (63 – 78)
1024	13 (0 – 16)	94 (94 – 94)	75 (63 – 78)
2048	3 (0 – 16)	113 (94 – 125)	88 (78 – 94)
4096	3 (0 – 16)	125 (109 – 156)	88 (78 – 94)
8192	3 (0 – 16)	163 (125 – 188)	125 (94 – 156)
16384	0 (0 – 0)	116 (109 – 125)	113 (94 – 156)

4 Conclusion

We have studied reactive proximity problems in graphs, giving a family of data structures for such problems. While we have focused on applications in geographic systems dealing with real-time data, the problem is primitive enough that it seems likely that it will arise in other domains of graph theory, such as network protocols. We would like to explore other applications in the future.

As we discussed in Sect. 3.1, a big factor in the runtime of any data structure based on separator hierarchies is the choice of separators. It may be of interest to compare the benefits of a simpler but lower-quality separator construction algorithm versus a slower and more complicated but higher-quality separator construction algorithm in future experiments.

References

1. Agarwal, P.K., Eppstein, D., Matoušek, J.: Dynamic half-space reporting, geometric optimization, and minimum spanning trees. In: 33rd Symposium Foundations of Computer Science (FOCS), pp. 80–89 (1992)
2. Arkin, E.M., Bae, S.W., Efrat, A., Okamoto, K., Mitchell, J.S., Polishchuk, V.: Geometric stable roommates. Inf. Process. Lett. **109**(4), 219–224 (2009). http://www.sciencedirect.com/science/article/pii/S0020019008003098

3. Aurenhammer, F.: Voronoi diagrams–a survey of a fundamental geometric data structure. ACM Comput. Surv. **23**(3), 345–405 (1991)
4. Buriol, L.S., Resende, M.G.C., Thorup, M.: Speeding up dynamic shortest-path algorithms. INFORMS J. Comput. **20**(2), 191–204 (2008)
5. Chan, E.P.F., Yang, Y.: Shortest path tree computation in dynamic graphs. IEEE Trans. Comput. **58**(4), 541–557 (2009)
6. Chan, T.M.: A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries. J. ACM **57**(3), 16:1–16:15 (2010)
7. Chung, F.R.K.: Separator theorems and their applications. In: Paths, flows, and VLSI-layout (Bonn, 1988), Algorithms Combin. vol. 9, pp. 17–34. Springer, Berlin (1990)
8. Clarkson, K.L.: Nearest-neighbor searching and metric space dimensions. In: Shakhnarovich, G., Darrell, T., Indyk, P. (eds.) Nearest-Neighbor Methods in Learning and Vision: Theory and Practice, pp. 15–59. MIT Press (2006). Chapter 2
9. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill, New York City (2001)
10. De Berg, M., Cheong, O., Van Kreveld, M., Overmars, M.: Computational Geometry: Introduction. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77974-2
11. Demetrescu, C., Goldberg, A.V., Johnson, D.S.: 9th DIMACS implementation challenge: shortest paths (2006). http://www.dis.uniroma1.it/~challenge9/
12. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM **51**(6), 968–992 (2004)
13. Djidjev, H.N., Pantziou, G.E., Zaroliagis, C.D.: On-line and dynamic algorithms for shortest path problems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 193–204. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59042-0_73
14. Dujmović, V., Eppstein, D., Wood, D.R.: Structure of graphs with locally restricted crossings. SIAM J. Discrete Math. **31**(2), 805–824 (2017)
15. Dvořák, Z., Norin, S.: Strongly sublinear separators and polynomial expansion. SIAM J. Discrete Math. **30**(2), 1095–1101 (2016)
16. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: 16th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 75–84 (1975)
17. Eppstein, D.: Dynamic Euclidean minimum spanning trees and extrema of binary functions. Discrete Comput. Geom. **13**(1), 111–122 (1995)
18. Eppstein, D.: Fast hierarchical clustering and other applications of dynamic closest pairs. J. Exp. Algorithmics **5** (2000)
19. Eppstein, D.: All maximal independent sets and dynamic dominance for sparse graphs. ACM Trans. Algorithms **5**(4), Article No. 38 (2009)
20. Eppstein, D.: Treetopes and their graphs. In: 27th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 969–984 (2016). http://dl.acm.org/citation.cfm?id=2884435.2884504
21. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Atallah, M.J. (ed.) Algorithms and Theory of Computation Handbook, pp. 9.1–9.28, 2nd edn. CRC Press(2010). http://www.info.uniroma2.it/~italiano/Papers/dyn-survey.ps.Z
22. Eppstein, D., Goodrich, M.T., Korkmaz, D., Mamano, N.: Defining equitable geographic districts in road networks via stable matching. In: 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (2017)

23. Eppstein, D., Goodrich, M.T., Sun, J.Z.: Skip quadtrees: dynamic data structures for multidimensional point sets. Int. J. Comput. Geom. Appl. **18**(1–2), 131–160 (2008)
24. Eppstein, D., Goodrich, M.T., Trott, L.: Going off-road: transversal complexity in road networks. In: 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 23–32 (2009)
25. Eppstein, D., Gupta, S.: Crossing patterns in nonplanar road networks. In: 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, September 2017
26. Erwig, M.: The graph Voronoi diagram with applications. Networks **36**(3), 156–163 (2000)
27. Frieze, A.M., Miller, G.L., Teng, S.H.: Separator based parallel divide and conquer in computational geometry. In: 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 420–429 (1992). http://doi.acm.org/10.1145/140901.141934
28. Frigioni, D., Italiano, G.F.: Dynamically switching vertices in planar graphs. Algorithmica **28**(1), 76–103 (2000)
29. Gilbert, J.R., Hutchinson, J.P., Tarjan, R.E.: A separator theorem for graphs of bounded genus. J. Algorithms **5**(3), 391–407 (1984)
30. Goodrich, M.T.: Planar separators and parallel polygon triangulation. J. Comput. Syst. Sci. **51**(3), 374–389 (1995)
31. Goodrich, M.T., Tamassia, R.: Algorithm Design and Applications, 1st edn. Wiley, Hoboken (2014)
32. Henzinger, M.R., Klein, P., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. J. Comput. Syst. Sci. **55**(1), 3–23 (1997)
33. Kaplan, H., Mulzer, W., Roditty, L., Seiferth, P., Sharir, M.: Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In: 28th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 2495–2504 (2017)
34. Kawarabayashi, K., Reed, B.: A separator theorem in minor-closed classes. In: 51st IEEE Symposium on Foundations of Computer Science (FOCS), pp. 153–162 (2010)
35. King, V.: Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: 40th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 81–89 (1999)
36. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM J. Appl. Math. **36**(2), 177–189 (1979)
37. Roditty, L., Zwick, U.: On dynamic shortest paths problems. Algorithmica **61**(2), 389–401 (2011)
38. Samet, H.: The design and analysis of spatial data structures. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading (1990)