The Graph Voronoi Diagram with Applications

Martin Erwig
FernUniversität Hagen
Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract

The Voronoi diagram is a famous structure of computational geometry. We show that there is a straightforward equivalent in graph theory which can be efficiently computed. In particular, we give two algorithms for the computation of graph Voronoi diagrams, prove a lower bound on the problem, and we identify cases where the algorithms presented are optimal. The space requirement of a graph Voronoi diagram is modest, since it needs no more space than the graph itself.

The investigation of graph Voronoi diagrams is motivated by many applications and problems on networks that can be easily solved with their help. This includes the computation of nearest facilities, all nearest neighbors and closest pairs, some kind of collision free moving, and anti-centers and closest points.

1 Introduction

The Voronoi diagram is a data structure extensively investigated in the domain of computational geometry [14]. Originally, it characterizes regions of proximity for a set of k sites in the plane where distance of points is defined by their Euclidean distance. Optimal algorithms exist to compute the Voronoi diagram in $\Theta(k \log k)$ time, and the Voronoi diagram can be represented in $\Theta(k)$ space. Although there have been proposed many extensions and generalizations [3] we know only of Mehlhorn [13] identifying this structure in the context of graphs. One reason for that may be that for graphs the above bounds cannot be achieved — the most general setting within the above bounds is characterized by so-called "nice metrics" [11, 12]. Nevertheless, it is worthwhile to investigate graph Voronoi diagrams because they efficiently support solutions to many network problems. We will consider these applications in Section 4.

In the following we shall drop the above mentioned time/space restrictions and consider the graph Voronoi diagram, which encodes proximity information in graphs based on shortest paths between nodes. Of course, in a directed graph with arbitrary (positive) edge weights the shortest path distance is, in general, not a metric at all. We will see that both space and running time bounds depend not only on the number of sites, but also on the number of nodes and edges of the underlying graph. In [13] the improvement of an algorithm for approximating minimal Steiner trees led to the discovery of this structure for undirected graphs. We will generalize the definition to directed graphs, and also take care of unreachable nodes, which were not considered by Mehlhorn. Concerning algorithms for constructing graph Voronoi diagrams, Mehlhorn just used an extension of Dijstra's shortest path algorithm. We refine this algorithm which leads to an asymptotic better behavior for large sets of Voronoi nodes. Moreover, we present an alternative algorithm which is more efficient in dense graphs, and we also demonstrate how both methods can be combined. Finally, we consider lower bounds on the problem, and characterize cases for which the presented algorithms are optimal.

Let $G = (N, E)$ be a directed graph with node set N and edge set $E \subseteq N \times N$. For a node v, the set of its successors is denoted by $suc(v)$, and $deg^+(v) = |suc(v)|$. Let $n = |N|$ and $m = |E|$. The mapping $c : E \to \mathrm{I\!R}^+$ assigns real-valued, positive costs to edges, and we use the term *network* for such a labeled graph. We write $c(v, w)$ to denote the cost of an edge $(v, w) \in E$. A path P of length ℓ from node s to node t is a sequence of nodes $\langle v_0, \ldots, v_\ell \rangle$ with $s = v_0$, $t = v_\ell$, and $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq \ell$. The cost of a path is defined as $c(P) = \sum_{i=1}^{\ell} c(v_{i-1}, v_i)$, and the cost of a shortest path between two nodes v and w is defined as $d(s, t) = \min\{c(P) \mid P \text{ is a path from } s \text{ to } t\}$. If there is no path from s to t, we define $d(s, t) = \infty$.

The following definitions are illustrated in Figure 1. In a directed graph $G = (N, E)$, the (*inward*) *Voronoi diagram* for a set of nodes $K = \{v_1, \ldots, v_k\} \subseteq N$ is a partition $\{N_1, \ldots, N_k, U\}$ of N so that

(i) for each node $v \in N_i$, $d(v, v_i) \leq d(v, v_j)$ for all $j \in \{1, \ldots, k\}$, and
(ii) U contains all nodes v with $d(v, v_i) = \infty$ for all $i \in \{1, \ldots, k\}$.

This means that the *inward* Voronoi diagram is based on the paths which lead *toward* the Voronoi nodes. Let $u = |U|$. The nodes of K are called *Voronoi nodes*, the N_i are called *Voronoi sets*, and U is called the set of *isolated* or *unreachable nodes*.

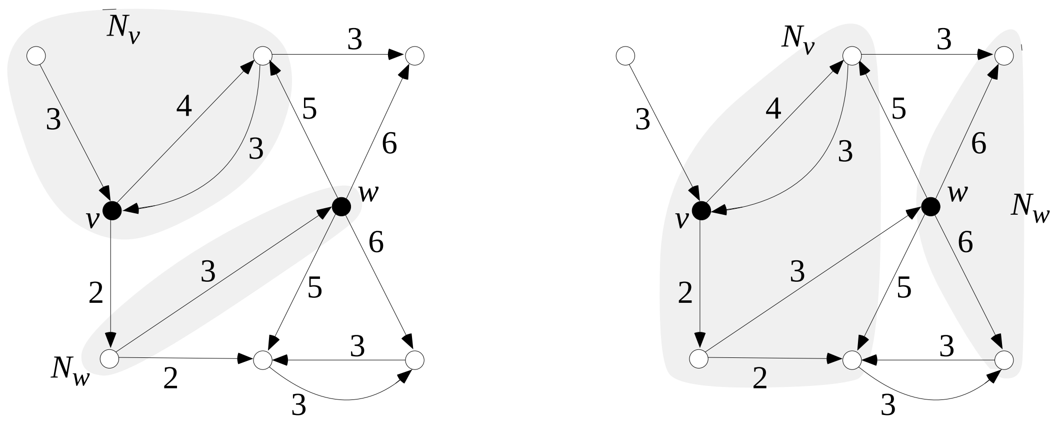

Figure 1: Inward and outward Voronoi diagram for the nodes v and w

For convenience we will sometimes write N_0 in place of U. For a fixed set K, we write $d(v)$ for $\min\{d(v, v_i) \mid v_i \in K\}$ and $\overline{d}(v)$ for $\min\{d(v_i, v) \mid v_i \in K\}$. The whole partition is denoted by $\mathit{Vor}(G, K)$, or just $\mathit{Vor}(K)$, if G is understood. The nearest Voronoi node of node v is given by

$$V(v) \;=\; \begin{cases} v_i & \text{if } \exists\, i \in \{1, \ldots, k\} : v \in N_i \\ \perp & \text{otherwise} \end{cases}$$

The use of $\perp$ expresses that V is undefined for isolated nodes.

Note that, in general, the Voronoi diagram is not uniquely defined, for example, if $d(v, v_i) = d(v, v_j)$ for $i \neq j$, then v may be assigned to either Voronoi set N_i or N_j. This is generally not a problem. If, for some reason, it should be required that the Voronoi diagram is uniquely defined, we can add a condition like

$$v \in N_i \implies i = \min\{j \mid d(v, v_i) = d(v, v_j)\}$$

which uses an (arbitrary) ordering among the Voronoi nodes to assign nodes unambiguously.

Next we define the set of *Voronoi bridges* E_B as the union of the sets E_{ij} with $i, j \in \{0, \ldots, k\}$ and $i \neq j$, where $E_{ij} = \{(v, w) \in E \mid v \in N_i \text{ and } w \in N_j\}$.

The *outward Voronoi diagram* is the dual of the inward one, that is, for each node $v \in N_i$ we have $d(v_i, v) \leq d(v_j, v)$ for all $j \in \{1, \ldots, k\}$, and U contains all nodes v with $d(v_i, v) = \infty$ for all $i \in \{1, \ldots, k\}$. The outward Voronoi node of node v is denoted by $\overline{V}(v)$. The terminology is chosen in analogy to the definition of medians and centers in [10]. Unless stated otherwise we will always refer to the inward Voronoi diagram.

In the following we shall ignore the trivial cases $k = 1$ where $\mathit{Vor}(K) = \{N - U, U\}$ and $k = n$ where $N = K = \{v_1, \ldots, v_n\}$ and $\mathit{Vor}(K) = \{\{v_1\}, \ldots, \{v_n\}, \varnothing\}$, that is, we assume $1 < k < n$. In Section 2 we prove some elementary properties of graph Voronoi diagrams. After that we consider their computation in Section 3. In Section 4 we show how to employ the graph Voronoi diagram in solving network problems. Conclusions follow in Section 5.

2 Basic Properties

We already noted that unlike its geometric counterpart the graph Voronoi diagram may contain some unreachable nodes. We can characterize the set U in part by the connectedness of G. Let $S_1, \ldots, S_\delta$ be the node sets of the strongly connected components of G, and let $G' = (N', E')$ be the induced directed acyclic graph defined by $N' = \{S_1, \ldots, S_\delta\}$ and $E' = \{(S_i, S_j) \mid \exists v \in S_i, w \in S_j : \; (v, w) \in E\}$.

A strong component S_i is called an (*inward*) *dead end with respect to* K iff the following two conditions hold:

(i) $K \cap S_i = \varnothing$ and
(ii) S_i has no successor in G' or all successors of S_i in G' are dead ends.

Outward dead ends have the dual definition. We have:

Lemma 1 *The set U of unreachable nodes of the Voronoi diagram $\mathit{Vor}(G, K)$ equals the union of all dead ends with respect to K.*

Proof. Consider a node w in a strong component S_i. First we show that all nodes of dead ends are contained in U. If no Voronoi node is contained in S_i, a path from w to a Voronoi node has to lead via an edge that induces an edge of E'. Now, if S_i has no outgoing edges in G', $d(w, v_j) = \infty$ for all $v_j \in K$ and thus $w \in U$. It follows by induction over the length of paths in G' that $w \in U$ if S_i has outgoing edges in G' which all lead to dead ends.

Assume on the other hand that S_i is not a dead end. If $K \cap S_i \neq \varnothing$, then there is a path from w to a Voronoi node in S_i and thus $w \notin U$. If $K \cap S_i = \varnothing$, then there must be a path in G' from S_i to a component S_j containing Voronoi nodes, for if this were not the case, all paths starting from S_i would end up in dead ends and by induction of the length of such paths it would follow that S_i itself is a dead end. Contradiction. So we have shown that U only contains nodes of dead ends. This completes the proof. □

Not surprisingly, the connectedness of G is related to the existence of unreachable nodes in the Voronoi diagram.

Corollary 1 *G is strongly connected* $\iff$ $U = \varnothing$. □

A simple representation of a graph Voronoi diagrams is an array that stores for each node its nearest Voronoi node (or $\perp$ for unreachable nodes). Concerning the storage requirement this means:

Lemma 2 *The graph Voronoi diagram can represented in $O(n)$ space.* □

Sometimes it is helpful to have the following additional information:

(i) for each node the distance d to its nearest Voronoi node,
(ii) for each Voronoi node v_i the corresponding Voronoi set N_i, and
(iii) for each Voronoi node v_i the Voronoi bridges E_{ij}

Clearly, (i) and (ii) require only $O(n)$ additional space. So we may assume in the following that this information is also available. The bridges may take up to $O(m)$ space, and if they are given, too, we call the corresponding Voronoi diagram *extended.*

Lemma 3 *The extended graph Voronoi diagram can represented in $O(m + n)$ space.* □

To store the underlying graph explicitly $O(m + n)$ space is needed, and thus the storage requirements for "normal" as well as for extended Voronoi diagrams lie within this bound. So from this point of view the above space bounds are tight.

The differences between inward and outward Voronoi diagrams vanish in undirected graphs; in particular, the respective Voronoi bridges are the same, that is, $E_{ij} = E_{ji}$ for all $1 \leq i, j \leq k$. This is an important precondition to the application of Voronoi diagrams in supporting the search for all nearest neighbors (see Section 4.2). Lemma 4 forms the basis for an appropriate algorithm. We say that two Voronoi nodes v_i and v_j are *direct neighbors* if $E_{ij} \neq \varnothing$. A *nearest neighbor* of a Voronoi node v_i is a Voronoi node $v_j \neq v_i$ with minimal $d(v_j, v_i)$.

Lemma 4 *In undirected graphs, a nearest neighbor of a Voronoi node is always one of its direct neighbors.*

Proof. Let v_j be any nearest neighbor of v_i, and assume v_j is not a direct neighbor of v_i. We show that this assumption leads to a contradiction. It follows from the assumption that the shortest path from v_j to v_i leads via at least two bridges because the shortest path has to enter and to leave a direct neighborhood of v_j, say N_h. Assume that the two bridges (v, w) and (x, y) lie on the shortest path from v_j to v_i and that $v \in N_j$, $y \in N_i$, and $x \in N_h$ where v_h is a direct neighbor of v_i. This situation is illustrated in Figure 2.

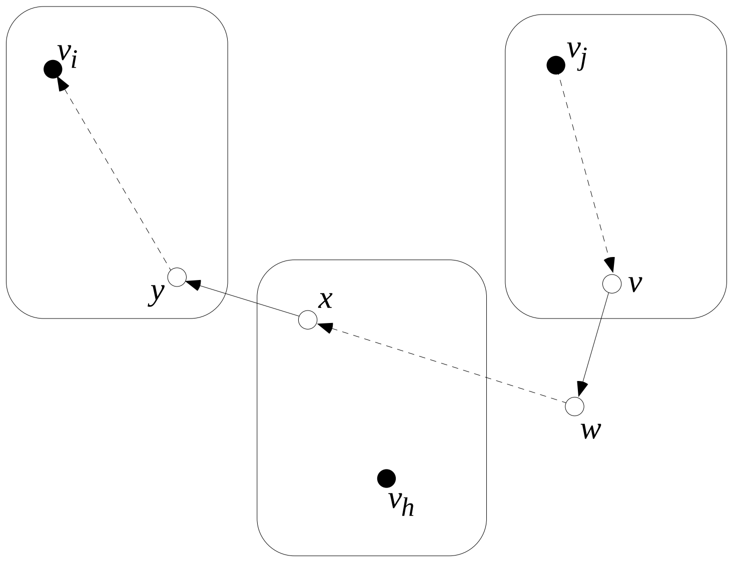

Figure 2: Illustrating the proof to Lemma 4

We know that

$$d(v_j, v_i) = d(v_j, v) + c(v, w) + d(w, x) + c(x, y) + d(y, v_i).$$

Moreover,

$$d(v_j, v_i) < d(v_h, x) + c(x, y) + d(y, v_i)$$

because otherwise v_h instead of v_j would be the nearest neighbor of v_i. Thus we get

$$d(v_j, v) + c(v, w) + d(w, x) < d(v_h, x)$$

which means that x is nearer to v_j than to v_h (because in undirected graphs we have, for example, $d(w, x) = d(x, w)$). But this contradicts our assumption $x \in N_h$. □

3 Computation of Graph Voronoi Diagrams

Let us first give a lower bound for the computation of graph Voronoi diagrams. We observe that $Vor(K)$ can be viewed as a mapping V from N to $K \cup \{\perp\}$. So with regard to the output of any algorithm, the trivial lower bound is $\Omega(n)$.

Now there are $(k+1)^{n-k}$ possibilities for V on $N-K$, and from an information theoretic point of view the complexity for computing one instance is $\Omega(\log(k+1)^{n-k}) = \Omega((n-k)\log k)$.

Putting these observations together we obtain

Theorem 1 *A lower bound for computing graph Voronoi diagrams is given by* $\Omega(\max(n, (n-k)\log k))$. □

Note that the lower bound is $\Omega(n \log n)$ for $k = O(n^{a/b})$, where a, b are constants with $b > a$. If $k = n - O(1)$ or $k = O(1)$, the lower bound is $\Omega(n)$.

Next we consider the computation of graph Voronoi diagrams.

3.1 The Parallel Dijkstra Algorithm

For the following algorithm we need the notion of the *reverse graph* G_r which is obtained from G by reversing the direction of all edges. Instead of constructing G_r explicitly we can interchange in the algorithm an access to successors of a node by an access to its predecessors and vice versa. Thus we can avoid (for the present) an $\Omega(m)$ lower bound for the algorithm. Of course, this presumes an appropriate storage of the graph.

Now, a first approach is to run a variant of Dijkstra's algorithm [1] on G_r with the Voronoi nodes as multiple sources. (Note that the outward Voronoi diagram can be computed in much the same way, except that the search has to be performed on G instead of G_r.)

We use the heap-based implementation of Dijkstra's algorithm, that is, we have available the operations *insert*(v, h), which inserts the node v into the heap h with the value $d(v)$, *decrease*(Δ, v, h), which decreases the value for v in h by Δ, *min*(h), which yields the minimal element of the heap h, and *deletemin*(h), which removes the minimal element from h. Using, for instance, Fibonacci heaps [8], all operations take $O(1)$ (amortized) time, except *deletemin*, which is $O(\log n)$. We assume that the heap h is arranged according to d. First, we perform the following initialization step:

```
init:
  for each $v \in N$ do
     if $v \in K$ then
        $d(v) := 0; V(v) := v; insert(v, h)$
     else
        $d(v) := \infty; V(v) := \perp$
```

We may assume without loss of generalization that the set membership can be tested in $O(1)$. This can be achieved by initially scanning K and marking all nodes in G appropriately.

Next, the nodes are expanded in the well-known manner until the heap is empty. When updating a node's tentative cost $d(w)$ where the update was "caused" by edge (v, w), we additionally set $V(w) := V(v)$. Thus, node expansion becomes

```
    expandnext:
        $v := deletemin(h)$
        mark $v$
        $scansuc(v)$
```

with:

```
    scansuc(v):
        for each outgoing edge $(v, w)$ with $w$ not marked do
            $\Delta := d(v) + c(v, w)$
            if $d(w) = \infty$ then
                $d(w) := \Delta; V(w) := V(v); insert(w, h)$
            if $d(w) < \infty$ and $\Delta < d(w)$ then
                $V(w) := V(v); decrease(\Delta, w, h)$
```

Altogether we have a variant of Dijkstra's algorithm which we will call "Parallel Dijkstra" since the shortest path trees starting from the Voronoi nodes grow rather simultaneously.

```
    Parallel Dijkstra:
        init
        while $h$ is not empty do
                expandnext
```

We observe that computing the graph Voronoi diagram is asymptotically not worse than the single source shortest path problem.

Theorem 2 *The Parallel Dijkstra algorithm computes the graph Voronoi diagram in a worst-case time of $O(m + n \log n)$.*

Proof. For correctness, suppose we add to G_r a new node s and edges (s, v_i) with $c(s, v_i) = 0$, for all $v_i \in K$. Consider a run of Dijkstra's algorithm starting with s. After the first step (that is, expansion of s) all v_i are on the heap with tentative costs of 0. So the remaining steps are equal to those of the parallel Dijkstra algorithm, which means that properties of Dijkstra's algorithm also hold for the parallel version. On termination, we obtain a shortest path tree in which each node's distance from s is equal to its distance from that v_i that lies on the shortest path from s. Therefore each node in a shortest path subtree rooted at a $v_i \in K$ is not further from v_i than from any other $v_j \in K - \{v_i\}$. Since V is appropriately updated whenever a node's shortest path tree membership may change, we conclude that for all reachable nodes the Voronoi diagram is computed correctly. In addition, the initialization step assures that V is set to $\perp$ for unreachable nodes.

Concerning running time, we observe that the initialization step takes $O(n)$ steps. The above run of Dijkstra's algorithm, and by analogy the parallel version, too, takes $O(m + n \log n)$ steps (if it is implemented using Fibonacci heaps). □

We can improve the algorithm if we take into account the following observation: the Voronoi nodes are always the first k nodes to be expanded. That is, we need not necessarily insert them into the heap. This is reflected in the following modified initialization step:

```
init':
    for each $v \in N$ do
        if $v \in K$ then
            $d(v) := 0; V(v) := v$; mark $v$
        else
            $d(v) := \infty; V(v) := \perp$
    for each $v \in K$ do
        $scansuc(v)$
```

Since the Voronoi nodes have to be marked to prevent them from being inserted into the heap, *scansuc* must follow in a second loop. For this version of the algorithm we obtain a slightly better bound because the number of *deletemin* operations reduces to $n-k$ and the initialization still takes $O(n)$ steps.

Corollary 2 *The parallel Dijkstra algorithm computes the graph Voronoi diagram in a worst-case time of $O(m+n+(n-k)\log(n-k))$.* □

Of course, this is an improvement only for sufficiently large k, and it affects the overall running time merely in sparse graphs.

Assume $m = O(n)$. Recall the lower bound from above, which is $\Omega(n \log n)$ for $k = O(n^{a/b})$, where a and b are constants with $b > a$. If $k = \Theta(n^{a/b})$, the Parallel Dijkstra algorithm runs in $O(n \log n)$, which is optimal. If $k = n - O(1)$, the lower bound is $\Omega(n)$ which, too, is actually achieved by Parallel Dijkstra. So we have

Theorem 3 *The Parallel Dijkstra algorithm is asymptotically optimal for $m = O(n)$ and $k = \Theta(n^{a/b})$, where a, b are constants and $b \geq a$.* □

3.2 A Direct Method

Concerning dense graphs, it is indeed possible to avoid the $\Omega(m)$ lower bound of the algorithm for large values of k. Instead of searching in G_r starting with the Voronoi nodes we can repeatedly build shortest path trees (in G) for nodes $v \in N - K$. Using Dijkstra's algorithm we stop when a Voronoi node v_i is taken from the heap. Then we can set $V(v) = v_i$ (and $V(w) = v_i$ for all w on the shortest path from v to v_i). These nodes are marked permanently. For all other nodes v that were expanded in the current run, we unmark v and set $d(v) = \infty$. We call this the *direct method*. Let $\overline{m}$ denote the number of edges (v, w) with $v \in N - K$. We have

Theorem 4 *The direct method computes the graph Voronoi diagram in a worst-case time of $O(n + (n-k)\overline{m} + (n-k)^2 \log(n-k))$.*

Proof. In an $O(n)$ initialization step we have to mark all nodes of K in G. Then, employing Dijkstra's algorithm we consider only edges (v, w) with $v \in N - K$ because we can stop after having taken a Voronoi node from the heap. We have to run the algorithm repeatedly each time starting with a node that was not marked permanently in any earlier run. Since in each run at least one node gets marked the algorithm is repeated at most $n-k$ times expanding no more than $n-k$ nodes each time. $\square$

The direct method is faster than Parallel Dijkstra at least if

$$(n-k)\overline{m} + (n-k)^2 \log(n-k) \quad < \quad m + (n-k)\log(n-k).$$

This can only be the case for $n-k = o(n/\sqrt{\log n})$ because otherwise $(n-k)^2\log(n-k)$ would be $\Omega(n^2)$. So if $m = n^a$ for $1 < a \leq 2$, and $(n-k) = n^b$, for $b < 1$, we get

$$\begin{aligned} n^b(\overline{m} + n^b b \log n) &< m + n^b b \log n \\ \overline{m} + n^b b \log n &< n^{a-b} + b \log n \\ \overline{m} &< n^{a-b} - (n^b - 1) b \log n. \end{aligned}$$

Thus we can combine both approaches in the following way: count $\overline{m}$ by visiting nodes of $N-K$ until $\overline{m} \geq n^{a-b} - (n^b-1)b\log n$ or all nodes of $N-K$ have been processed. In the latter case perform the direct method, otherwise use Parallel Dijkstra. We collect our results in

Corollary 3 *The graph Voronoi diagram can be computed in a worst-case time of* $O(n + \min(m + (n-k)\log(n-k), (n-k)\overline{m} + (n-k)^2\log(n-k)))$. $\square$

3.3 Updating Voronoi Diagrams

We can distinguish two kinds of updates that require a rearrangement of the Voronoi diagram:

(i) altering the graph, that is, adding/dropping nodes and edges, and changing c, or
(ii) altering the set of Voronoi nodes

The first case reduces to the problem of updating shortest path trees according to changes of the underlying graph. This has already been investigated elsewhere, see discussion and references in [5] on page 284. So here we only consider the latter case.

Adding a new Voronoi node v_{k+1} to K can be done as follows. We insert the new node into a heap (with cost 0), and then we repeatedly expand nodes using Dijkstra's algorithm (in G_r for the inward diagram) where we insert only those successor nodes into the heap which get assigned a smaller distance than in the old Voronoi diagram. This means that these nodes are nearer to v_{k+1} than to their "old" Voronoi nodes.

The running time of one such update is $O(m_{k+1} + n_{k+1}\log n_{k+1})$ with $n_{k+1} = |N_{k+1}|$ and m_{k+1} denoting the number of edges (v,w) with $v \in N_{k+1}$. Unfortunately, this can become as much effort as constructing the whole diagram from scratch, that is, $n_{k+1} = O(n)$ and $m_{k+1} = O(m)$.

To remove a Voronoi node v_i from K we first have to address the fact that nodes of the Voronoi set N_i may now become unreachable. Therefore, all d- and V-values for nodes of N_i are reset to ∞ and $\perp$, respectively. After that we scan all Voronoi bridges of v_i putting the corresponding nodes of N_i into a heap. Finally, repeated node expansion (in G_r for the inward diagram) splits N_i among the bordering Voronoi sets (and U).

```
*Drop Voronoi Node*($v_i$):
    **for each** $v \in N_i$ **do**
        $d(v) := \infty; V(v) := \perp$
    **for each** $(w, v) \in E_{ij}$ **do**
        $\Delta := d(v) + c(w, v)$
        **if** $d(w) = \infty$ **then**
            $d(w) := \Delta; V(w) := V(v); insert(w, h)$
        **if** $d(w) < \infty$ **and** $\Delta < d(w)$ **then**
            $V(w) := V(v); decrease(\Delta, w, h)$
    Set all $E_{ji}$ to $\varnothing$
    **while** $h$ is not empty **do**
        *expandnext*
```

Note carefully that the **for**-loops refer to G whereas the operation *expandnext* operates on G_r.

Here we made use of the extended Voronoi diagram to obtain the Voronoi bridges. If it is not available, we can reconstruct all relevant bridges by a run of Dijkstra's algorithm starting with v_i (similar to the one performed when adding Voronoi nodes) to get the border vertices of the Voronoi region N_i. As with adding Voronoi nodes, the running time may become $O(m + n \log n)$ in the worst case.

Note that the sets of bridges E_{ij} are no longer relevant after node deletion because v_i is not a Voronoi node anymore. The other bridges of adjacent Voronoi regions are deleted just after the second **for**-loop.

Concerning the outward diagram, the second loop must be simply changed to:

```
**for each** $(v, w) \in E_{ji}$ **do**
    $\Delta := \overline{d}(v) + c(v, w)$
    ...
```

Moreover, we have to use $\overline{d}$ instead of d in all other parts of the algorithm, and *expandnext* must operate on G.

4 Applications

The idea of the graph Voronoi data structure came up during the investigation of queries typically found in databases storing network information [7]. We shall first consider the counterparts of some standard problems in computational geometry that can be solved with the help of Voronoi diagrams, namely, nearest neighbor, all nearest neighbors, closest pair, and collision-free moving. In addition, we consider anti-centers, a graph-theoretic concept naturally arising from graph Voronoi diagrams.

To illustrate the following applications we will assume that G models a transportation network and that K is a set of nodes where facilities, such as, fire stations, hospitals, post offices, or shopping malls, are located.

4.1 Nearest Facilities

The nearest facility of a query point v is that Voronoi node to which the shortest path from v has minimal cost. The following information may be of interest:

(i) Which Voronoi node v_i is nearest to v ?
(ii) How far is v_i from v ?
(iii) What is the shortest path from v to v_i ?

As an example suppose an accident happens at node v. Then we seek the shortest path to the nearest hospital.

Given a Voronoi diagram $\mathit{Vor}(G, K)$, tasks (i) and (ii) can be performed in $O(1)$ by simply looking up the values of $V(v)$ and $d(v)$, respectively. If the shortest path from v to $V(v)$ is of length ℓ, reconstructing that path can be performed node by node: suppose that we have already found a shortest path from v to x. Then the next node on a shortest path to $V(v)$ is any successor w of x with $c(x,w)+d(w) = d(x)$. This takes $\sum_{i=0}^{\ell-1} \mathit{deg}^+(v_i)$ steps, and in the worst case this may require $O(n\ell) = O(n^2)$ time, but note that in any case this procedure is faster than, for example, Dijkstra's algorithm. In particular, for sparse graphs the bound is expected to be $O(\ell)$ since there are only constantly many successors on the average. Because spatial networks are mainly sparse, this means that in our example the shortest path can be found in linear time. Moreover, we can achieve $O(\ell)$ searching time in arbitrary networksif we store in addition to the Voronoi diagram those edges belonging to the shortest path forests that have been built during the Voronoi search. This needs only $O(n)$ additional space, and the shortest path can then easily be obtained by tracing back these edges in the appropriate shortest path tree.

Without a graph Voronoi diagram all three tasks require the computation of the shortest path from v to $V(v)$ which takes $O(m+n \log n)$ when using Dijkstra's algorithm. This shows that on the one hand (pre-) computing the Voronoi diagram is asymptotically not worse than computing nearest facilities directly and that on the other hand it pays off especially in cases when multiple nearest facility queries are posed (for the same set K, of course).

Related to nearest facilities is the problem of "constrained routing". In the simplest case, a path is to be found from s to t that passes through at least one node with a specific property (or equivalently, through any node from a specific subset of nodes) and is otherwise as short as possible. Of course, a shortest path from s to t does not have the desired property in general. In some cases the graph Voronoi diagram can help finding a solution: assume we have computed both (inward and outward) Voronoi diagrams for the subset of nodes to be visited. Now if $V(s) = \overline{V}(t)$, the optimal constrained route is obtained by concatenating the shortest paths from s to $V(s)$ and from $V(s)$ to t which can be reconstructed in linear time. This is certainly only of use when s and t lie in the "neighborhood" of the same Voronoi node which is probably to occur whenever facilities are few and s and t are not located to far from one another.

4.2 All Nearest Neighbors and Closest Pair

A first approach to obtain all pairs of nearest Voronoi nodes is to perform Dijkstra's algorithm from each Voronoi node stopping when another Voronoi node is expanded.

Clearly, this will take $O(km + kn \log n)$ steps in the worst-case. In general, there seems to be an improvement employing the graph Voronoi diagram only for undirected graphs because in that case the inward and outward Voronoi diagrams coincide, which means that the Voronoi bridges are the same.

Thus, in undirected graphs we can scan all Voronoi bridges maintaining two arrays of length k, C and C_d, which record for each Voronoi node the currently closest neighbor and the cost of the shortest path to it as follows.

```
All Nearest Neighbors:
    for each $i \in \{1, \ldots, k\}$ do
        $C(i) := \perp; C_d(i) = \infty$
    for each $(v, w) \in E_B$ do
        $D := d(v) + c(v, w) + d(w)$
        if $D < C_d(V(v))$ then $C(V(v)) := V(w); C_d(V(v)) := D$
        if $D < C_d(V(w))$ then $C(V(w)) := V(v); C_d(V(w)) := D$
```

This method presumes the existence of a bridge between any two nearest Voronoi nodes. Fortunately, this assumption is justified by Lemma 4 given in Section 2.

The initialization takes $O(k)$ steps, and the scan of the Voronoi bridges takes $O(|E_B|)$ step. Altogether, all nearest neighbor can be found in $O(k + |E_B|) = O(k + m)$ steps. In sparse graphs this becomes $O(k + n)$, and for regularly shaped graphs (that is, sparse spatial graphs in whose embeddings the square of the diameter equals the space occupied, see [6]) it is expected to be $O(k + \sqrt{n})$.

The closest pair of Voronoi nodes can be either obtained as the minimum of the above all nearest neighbors arrays, or it can be computed by a modified version of Parallel Dijkstra where we allow each non-Voronoi node to be expanded more than once, but at most k-times (via paths from different Voronoi nodes). We stop when the first Voronoi node is expanded. This node together with that Voronoi at which the shortest path tree is rooted is the closest pair. Note that this algorithm will work for directed graphs, too.

For an application suppose the Voronoi nodes are shopping malls. Then the closest pair indicates those two malls with expected maximum competition whereas the list of all nearest neighbors gives for each mall the greatest competitor.

4.3 Collision-Free Moving

In the Euclidean plane the edges of the Voronoi diagram for a set of obstacle sites define locally maximal distances to the obstacles. Moving along the edges is a good policy to avoid collisions [2]. These edges correspond to bridges in the graph model, and in general, there are no edges (of the graph) between bridges (that is, endpoints of bridges). Hence, keeping always a maximum distance from the sites makes no sense because we cannot always move directly from one bridge to another. Indeed from an application point of view, the task of keeping always a predefined minimum/maximum distance seems to be more appropriate.

Imagine that K is a set of water purification units and we are planning a transport of polluting goods. To avoid a pollution in case of an accident we have to keep a minimum distance of D from each of the sites. This is supported by the Voronoi diagram in an easy way by deleting all nodes from the graph with $d(v) < D$ and then searching the shortest path in the modified graph.[1] If only few queries are posed and the paths to be found are small with respect to n, we could instead consider during the search only nodes with $d(v) \geq D$.

As an example for not exceeding a maximum distance consider the transport of a patient from one hospital to another. The distance to the currently nearest hospital may never be more than D so that in case of emergency we can quickly reach an operating theater.

4.4 Anti-Centers and Furthest Points

Centers are a well-known concept of graph theory [4]. The (*inward*) *center* of a graph is defined as the set of nodes with minimum eccentricity where the (*inward*) *eccentricity* of a node v is defined as $\max\{d(w,v) \mid w \in N\}$ [10]. The *inverse center* [9], or *periphery* [4], is defined as the set of nodes with maximum eccentricity.

In addition, we define the following kind of centers. In a directed graph $G = (N, E)$, the *inward* (*outward*) *anti-center* with respect to K is the set of nodes $v \in N$ with maximum $d(v)$ ($\overline{d}(v)$). This means that anti-centers are nodes which are farthest away from K.

Finding the anti-center has the following applications: Suppose that K is the set of nodes at which police stations are located. Since a clever bank robber wants to avoid the police she seeks for her robbery that node (at which, of course, a bank is located) which is farthest away from K. There is another, less criminal, example where the Voronoi nodes are shopping malls: a chain of stores plans to build a new shopping mall, and in order to minimize competition, a node in the graph is sought which is located as far as possible from the set K of already existing shopping malls.

Given a Voronoi diagram, the anti-center can be obtained in time $O(n)$ by looking up all $d/\overline{d}$-values. If the Voronoi diagram is not given, the anti-center can be computed by running the Parallel Dijkstra algorithm and pushing all expanded nodes onto a stack. After the algorithm has finished pop off all nodes from the stack that have the same d-value as the last node taken from the heap. The anti-center is just the set of these nodes.

The anti-center must not be confused with the well-known furthest point query, which asks for the Voronoi node which is farthest from a query node v. This may be obtained by executing Dijkstra's algorithm starting with v and stopping when the last Voronoi node is taken from the heap. The anti-center rather corresponds to the largest empty circle query [15].

5 Conclusions

We have defined the Voronoi diagram for graphs, and we have shown that (i) it can be efficiently computed, (ii) it needs asymptotically no more space than the graph itself, and

[1]This is correct only if the polluting goods move along the network to the purification unit (for instance, in a river or canal network).

(iii) it can be used for quite many interesting applications. We feel that the significance and simplicity of this concept stems from the fact that the Voronoi diagram is a natural extension of the shortest path problem (multiple sources plus concurrent search) and that proximity is a central concept in many graph applications.

From another point of view the present work generalizes Euclidean geometry to what could be called “graph geometry”. Beyond Voronoi diagrams it would be interesting to investigate other concepts and algorithms from computational geometry in the context of graphs (planar graphs may be an interesting intermediate stage). For instance, an obvious definition for the convex hull with respect to a set of nodes K is the set of nodes that are contained in any shortest path between any two nodes of K. There are some operations that are not interesting at all, for example, region intersection reduces to the intersection of node sets. But there are also some less obvious relationships, for instance, a polyline corresponds to a constrained shortest path, or a visibility region corresponds to a set of nodes to which the shortest path leads via a specified set of nodes.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley (1983).

[2] H. Alt and C. K. Yap. Algorithmic Aspects of Motion Planning: A Tutorial (Part 2). *Algorithms Reviews* **1** (1990) 61–77.

[3] F. Aurenhammer. Voronoi Diagrams – A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys* **23** (1991) 345–405.

[4] F. Buckley and F. Harary. *Distance in Graphs.* Addison-Wesley (1989).

[5] N. Deo and C. Pang. Shortest-Path Algorithms: Taxonomy and Annotation. *Networks* **14** (1984) 275–323.

[6] M. Erwig. Encoding Shortest Paths in Spatial Networks. *Networks* **26** (1995) 291–303.

[7] M. Erwig and R. H. Güting. Explicit Graphs in a Functional Model for Spatial Databases. *IEEE Transactions on Knowledge and Data Engineering* **5** (6) (1994) 787–804.

[8] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM* **34** (1987) 596–615.

[9] S. L. Hakimi. Optimum Distribution of Switching Centers in a Communication Network and Some Related Graph Theoretic Problems. *Operations Research* **13** (1965) 462–475.

[10] G. Y. Handler and P. B. Mirchandani. *Location on Networks: Theory and Algorithms.* MIT Press (1979).

[11] R. Klein. *Concrete and Abstract Voronoi Diagrams.* LNCS 400. Springer-Verlag (1989).

[12] R. Klein and D. Wood. Voronoi Diagrams Based on General Metrics in the Plane. *Symp. on Theoretical Aspects of Computer Science* (1988) 281–291.

[13] K. Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters* **27** (1988) 125–128.

[14] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag (1985).

[15] M. I. Shamos and D. Hoey. Closest Point Problems. *IEEE Symp. on Foundations of Computer Science* (1975) 151–162.