SHORTEST SECURE PATH IN A VORONOI DIAGRAM

BY

RAJGOPAL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Advisor:

Professor Sariel Har-Peled

ABSTRACT

We investigate the problem of computing the shortest secure path in a Voronoi diagram. Here, a path is secure if it is a sequence of touching Voronoi cells, where each Voronoi cell in the path has a uniform cost of being secured. Importantly, we allow inserting new sites, which in some cases leads to significantly shorter paths. We present an $O(n \log n)$ time algorithm for solving this problem in the plane, which uses a dynamic additive weighted Voronoi diagram to compute this path. The algorithm is an interesting combination of the continuous and discrete Dijkstra algorithms. We also implemented the algorithm using `CGAL`.

To my parents, for their love and support, and Big B, for the music.

ACKNOWLEDGMENTS

I would like to thank my advisor for giving me the opportunity to work on this wonderful problem and for his patience and support throughout the many difficulties, technical and otherwise, that arose in the time spent on it. I don't think I've ever come away from meeting him without feeling that the time to resolving what I didn't understand had been cut in half or better.

I would also like to thank all the amazing people that make `CGAL` (the Computational Geometry Algorithms Library) what it is and allow us to actually implement algorithms such as the one in this thesis. I was once one content to see things solely in the mind's eye, but seeing them in reality was actually far more satisfying.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Motivation. Consider a facility in an environment where other facilities exist. A client can communicate safely only with its nearest facility. The region where one can safely communicate with the facility, is the ***Voronoi cell*** of this facility. Given two facilities s and t, consider the problem of creating a safe corridor, where one can safely move from s and t while still being able to communicate safely with both of them. This might require inserting new middle facilities, such that the union of the new Voronoi cells (together with the Voronoi cells of s and t), form a connected set. The natural question is where and how many sites one needs to use/insert to establish such a reliable connection, see Figure 1.1.

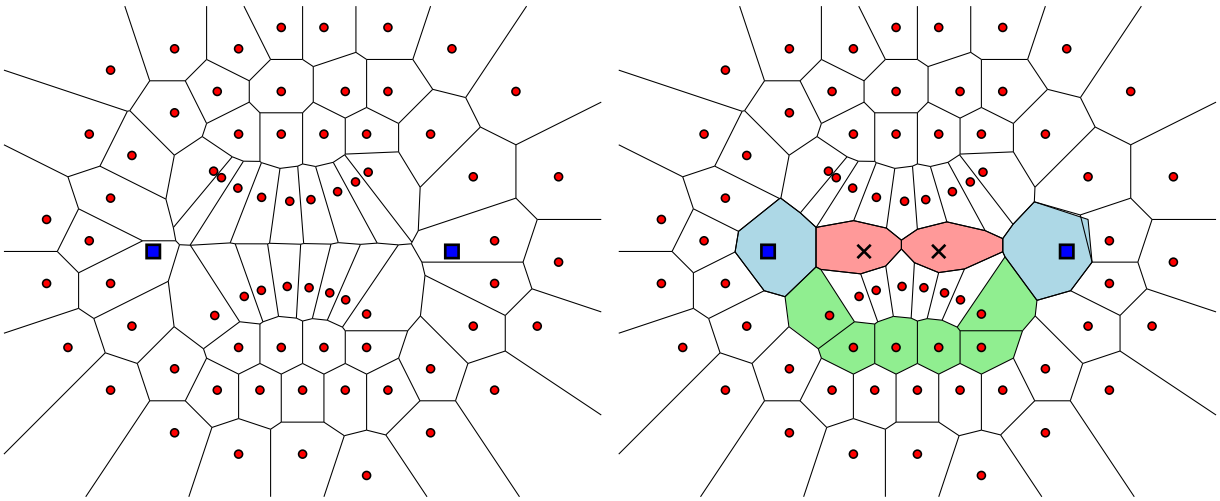


Figure 1.1: Inserting two middle sites is enough. Here we insert both the two endpoints into the diagram, and two middle sites (i.e., the two red cells). Note, that one can secure an existing site, instead of inserting a new one – this has the same cost. In this specific example, if one uses only existing cells, the price is six (the green cells).

Formal problem statement. Let P be a set of n points in the plane, and let $\mathcal{V}(P)$ denote its Voronoi diagram. Two points s, t in the plane can directly communicate *safely*, if s and t are "close" to each other. Formally, we require that the cells of s and t are adjacent in the Voronoi diagram of $P \cup \{s, t\}$ – geometrically, this corresponds to the existence of a disk that contains s and t, and no other points of P. Naturally, most points can not communicate safely. To overcome this, one can insert a set of points $Q = \{q_1, \ldots, q_\nu\}$ into the P, such that $q_0 = s$, $q_\nu = t$, and in the new diagram $\mathcal{V}(P \cup Q)$ the point q_i can safely communicate with q_{i+1}, for all i. It is natural to ask for the minimum size set Q, such that s and t can communicate safely, via $\nu - 1$ hops. Note, that a site q_i might be an existing site – this can

be interpreted as securing this site (this cost the same as inserting a new site).

A naive approach is to insert s and t into the initial Voronoi diagram, and compute the shortest path between them in the resulting dual graph (i.e., the Delaunay triangulation of $P \cup \{s, t\}$) – this corresponds to adding all the intermediate nodes in this path to Q. However, there are natural scenarios, see Figure 1.1, where allowing the sites to be placed arbitrarily in the plane significantly reduces the number of sites needed. As mentioned earlier, the price of securing an existing site, or introducing a new site is the same.

The task at hand is thus to compute the maximal regions $\mathcal{R}_i$ that can be reached by inserting i sites, from a starting site s, for all i. The region $\mathcal{R}_i$ is a closed connected set that might have holes. Once the ith level and the Voronoi cell of the target site t share a boundary, we will have obtained the smallest number of points that need to be inserted to secure a pathway between the two points. One could then also determine the actual locations of the points to be inserted.

Our results. We describe how the region of points that can be reached by (say) t insertions looks like, and to compute it efficiently. We then describe an $O(n \log n)$ time algorithm for computing the shortest such path between two points, using a process that is a variant of the Dijkstra algorithm. We implemented the algorithm using CGAL, and provide the source code.

Related results. The basic algorithm is a variant of continuous Dijkstra, a technique that was used for shortest path algorithms among obstacles, and for shortest path on polyhedral surfaces in three dimensions [1]. The basic process of inserting points, is similar in nature to Delaunay refinement [2]. The new algorithm can be viewed as a combination of these two techniques.

CHAPTER 2: THE ALGORITHM

Notations. For a point p and a radius r, let $\bigcirc(p, r)$ denote the disk of radius r centered at p. For a set of objects $\mathcal{X}$, in the plane, let $\mathcal{V}(\mathcal{X})$ denote the Voronoi diagram of $\mathcal{X}$. For an object x, let $\text{cell}_x(\mathcal{X})$ denote the cell of x in the Voronoi diagram of $\{x\} \cup \mathcal{X}$.

2.1 UNDERSTANDING THE PROBLEM: THE FLOWERING PROCESS

As an example, consider a point set formed by a hexagonal lattice – see Figure 2.1. Let $q_0 = s$ be a starting site (which is one of the points of the lattice), and assume our purpose is to reach a site $q_\nu = t \in P$. The site s can communicate directly with all the sites that are adjacent to its Voronoi cell $R_0 = \text{cell}_s(P)$, without requiring the insertion of any middle sites.

If we insert a new middle site q_1, then its cell must touch the cell of R_0. Consider such a location p for a middle site. For the new site of p to be adjacent to R_0, after p is inserted, there must be a point $v \in R_0$, such that v is as close (or closer) to p than it is to q_0. This region of influence, where v can be "occupied", is exactly the disk $\bigcirc(v, \|s - q_0\|)$. See Figure 2.1. As such, the allowable region to insert this first site, is

$$U_1 = \cup_{v \in R_0} \bigcirc\big(v, \|v - q_0\|\big). \tag{2.1}$$

(Namely, sites inserted outside U_1 are not going to be adjacent to R_0 in the new Voronoi diagram.) The set U_1 is a union of disks – in this specific case, it is the union of disks placed at vertices of the original Voronoi diagram, see Figure 2.1.

Let $P_1 = P \setminus U_1$. The region that can potentially be covered by such a single insertion of a site into U_1, belong to the region

$$R_1 = \bigcup_{p \in U_1} \text{cell}_p(P_1), \tag{2.2}$$

The set R_1 contains all the points, in the plane, that are closer to U_1, than to any point in P_1. Namely, $R_1 = \text{cell}_{U_1}(P_1)$. Any site $u \in P_1$ that its Voronoi cell is adjacent to R_1 in $\mathcal{V}(\{U_1\} \cup P_1)$, can communicate with q_0 via the insertion of a single site.

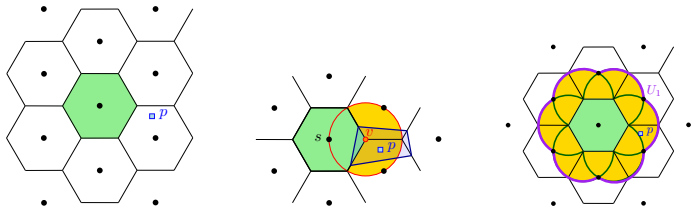


Figure 2.1: The hexagonal case.

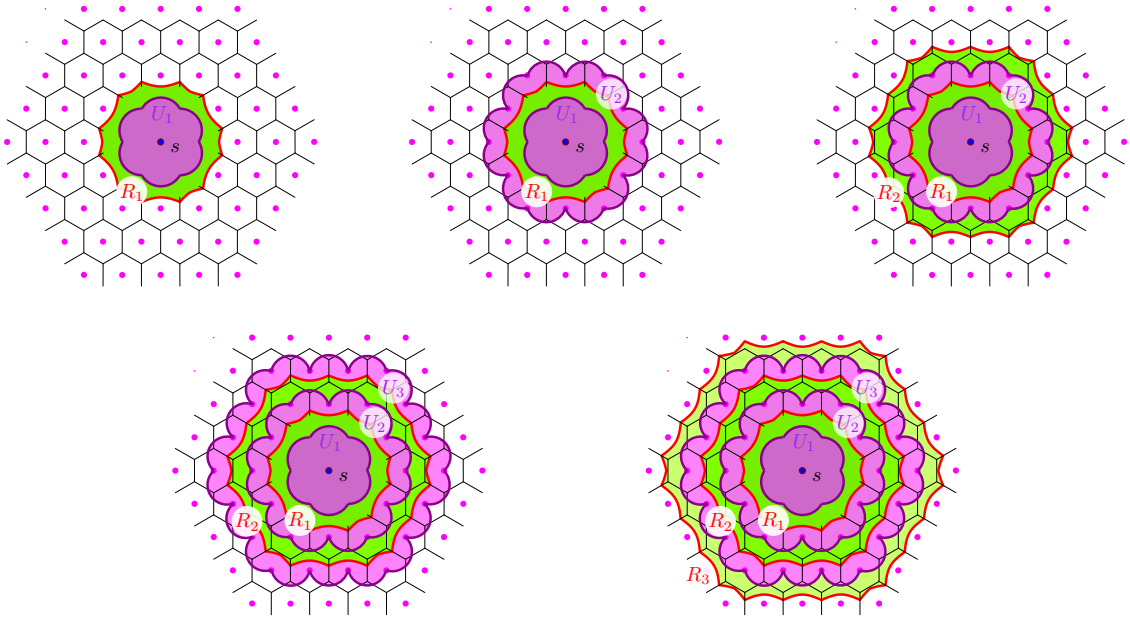


Figure 2.2: The flowering process of how the shortest path region grows.

The set R_1 is the Voronoi cell of U_1 in $\mathcal{V}(P_1 \cup \{U_1\})$. Crucially, the outer boundary of the region U_1 can be described by the boundary of the union of only a finite number of disks. These disks are precisely those placed at the Voronoi vertices of R_0. Indeed, to characterize the boundary of U_1 we first need only consider the disks $\bullet(v, \|v - q_0\|)$ where v lies on the boundary of R_0, ignoring disks in the interior of R_0. Then it is clear that among these, only the disks on the Voronoi vertices of R_0, as illustrated in figure Figure 2.3, are required to describe the boundary of U_1. A brief argument for this is as follows: since Voronoi edges are bisectors of adjacent sites, any disk with its center on an edge (henceforth just called "a disk on an edge") intersects the disks at the endpoints of the edge (i.e. the Voronoi vertices) exactly at the two sites defining the edge. Since two circles intersect at most at 2 points, the disks on the edges do not intersect the disks on the vertices anywhere else. Consequently, for

any disk on an edge, the two arcs of the disk cut by the sites defining the edge completely lie inside the disks at the vertices. Taking the union over all edges, the boundary of U_1 is defined completely by the boundary of the disks at the vertices of R_0.

As such, we may treat U_1 simply as the union of disks at the finitely many vertices of R_0. The boundary of U_1 is all that is necessary to accurately compute the boundary of R_1 (which may be viewed as a wavefront computed by the algorithm). R_1 is then the union of Voronoi cells in a Voronoi diagram of disks and points. This underlying diagram is therefore an additive weighted Voronoi diagram, and its boundary edges are portions of hyperbolas and straight segments. See Figure 2.2.

One concern with inserting only the bounding disks to function as the whole region is the introduction of artifacts such as holes or "pockets" that require careful treatment. However, the existing tools for additive Voronoi diagrams in CGAL allow them to be handled with relative ease.

Another potential concern is when the region R_0 contains a vertex at infinity, i.e. the cell is unbounded because it is at the edge of the diagram. In this case the circle through the point at infinity degenerates into two rays which we have to include in the Voronoi diagram. We avoid this issue by adding bounding points around the input sites to make the input points be in the interior. Another plausible route would have been to add support for rays to the AW-Voronoi library in CGAL.

The finiteness argument for the insertion of disks works as long as disks on edges are covered by disks at the vertices. When one site is itself a disk and the other is a point, disks on the edges and disks at the vertices still intersect at the point and then somewhere on the other side of the edge. The coverage of the disks on the edges is no longer complete, and this creates the previously mentioned issue of pockets, as in Figure 2.4. However, the disks at the vertices of the ith reachable region still completely cover the boundary of the $(i+1)$th occupied region, allowing us to continue working with an additive weighted Voronoi diagram throughout.

Formally, let the set R_i be the set reachable by inserting i middle sites starting at s. The set U_{i+1} is the region where one can insert an $(i+1)$th point, q_{i+1}, such that one can form a connected, safe chain back to s. By the discussion above, U_{i+1} can be treated as the union of finitely many disks at the vertices of R_i, with some additional care. Let $P_{i+1} = P \setminus U_i$. The $(i+1)$th *occupied* region is

$$U_{i+1} = \cup_{v \in R_i} \bullet \big(v, \mathsf{d}(v, P \setminus R_i)\big), \tag{2.3}$$

where $\mathsf{d}(v, P) = \min_{p \in P} \|v - p\|$. And the $(i+1)$th ***reachable*** region is

$$R_{i+1} = \bigcup_{p \in U_{i+1}} \text{cell}_p(P), \tag{2.4}$$

As soon as the region R_{i+1} shares a boundary with the Voronoi cell of t, one can stop, and reconstruct the safe path.

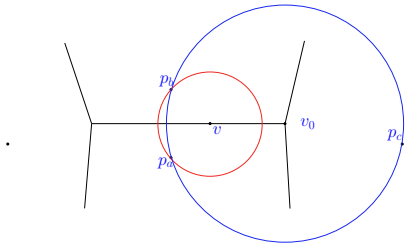


Figure 2.3: Why circles at vertices are enough. Here, suppose we have a portion of a Voronoi diagram with sites p_a, p_b and p_c. Consider a vertex v on the edge between p_a and p_b. The red disk centered at v through p_a also passes through p_b, as the edge is a bisector. Thus the arc of red disk from p_a to p_b on the same side of v_0 lies entirely within the blue disk. The remaining part of the red disk would be covered by a disk at the other endpoint of the edge. The covering argument would still work when p_a is a circle and p_b is a point, and the edge is a hyperbolic arc, but in this case we can only guarantee covering the portion of the disk on the same side of the arc as the point, p_b. The potentially problematic case when both p_a and p_b are circles with positive radius never arises in the course of the algorithm.

2.2 THE ALGORITHM

2.2.1 The basic algorithm.

The input is a set P of n points in the plane, and two point $s, t \in P$. The algorithm initially computes the Voronoi diagram of P, and has a queue $\mathcal{Q}$ of sites, which is initially set to s. A site here is a disk (the initial point is a disk of radius 0, naturally).

The algorithm works in rounds. In each round, it extracts all the sites in the queue, and inserts them into the current additive weighted Voronoi diagram – these are the outer disks, whose outer boundary forms the boundary of U_i in the ith round. Next, the algorithm scans all the inserted sites, and looks at the adjacent Voronoi vertices of their cells. All of the Voronoi vertices that are outside U_i (how to check for this condition we describe below), are inserted into the queue to be handled in the next round.

Once the wavefront reaches t, the processes stops – or alternatively, we compute this diagram till the whole plane is covered, and preprocess the resulting map for point location.

What disks to insert. What the algorithm does is that it inserts the disks of Eq. (2.3). As discussed before, one needs to insert only the disks that have their center on the boundary of R_i. This boundary is a union of hyperbolic segments. For simplicity and efficiency, we insert only the vertices of these hyperbolic edges. As illustrated in Figure 2.4, there are a few different kinds of vertices that appear. For one, the sites of the original unweighted Voronoi diagram will themselves lie along hyperbolic edges of the additive Voronoi diagram every time they are part of the boundary of some region U_i. However, sites of the original diagram are rarely Voronoi vertices of cells in the additive diagram.

The discussion of the earlier section implies that all Voronoi vertices of type "A" as in Figure 2.4 are necessary to correctly generate the regions U_i.

However, the insertion of circles causes edges to be created in the additive diagram along bisectors of adjacent circles. These edges intersect the wavefront at points of type "B" as in Figure 2.4. They are equidistant from one undiscovered site of the original diagram and two adjacent inserted disks. As such, the largest empty circles at these vertices do not actually participate in defining any boundary U_i, except when these points are also points of type "A" in degenerate cases. Any choice to include or exclude these points will have no effect on the correctness of the algorithm. Not inserting these points would reduce the complexity of the additive Voronoi diagram at every wavefront and may be more efficient in practice. However, the improvement would not be major and must be weighed against the check to distinguish between Voronoi vertex types on the wavefront. In an implementation, it is usually easier in practice to include all Voronoi vertices on the boundary of the regions R_i without concerning ourselves with the distinction.

Finally, we have the previously anticipated issue of "pockets" that are behind the front. Since these pockets can not contribute to the front, and they are shallow – we leave them behind. Such a pocket is marked in Figure 2.4.

To correctly leave the pockets behind, the algorithm remembers for each site inserted the layer (i.e., generation) of the propagation it came from. As such, when inspecting a Voronoi vertex, the algorithm knows the sites that gave rise to it, and their generation. All sites that are from two or more generations ago are not inserted, thus blocking the wavefront from propagating backwards.

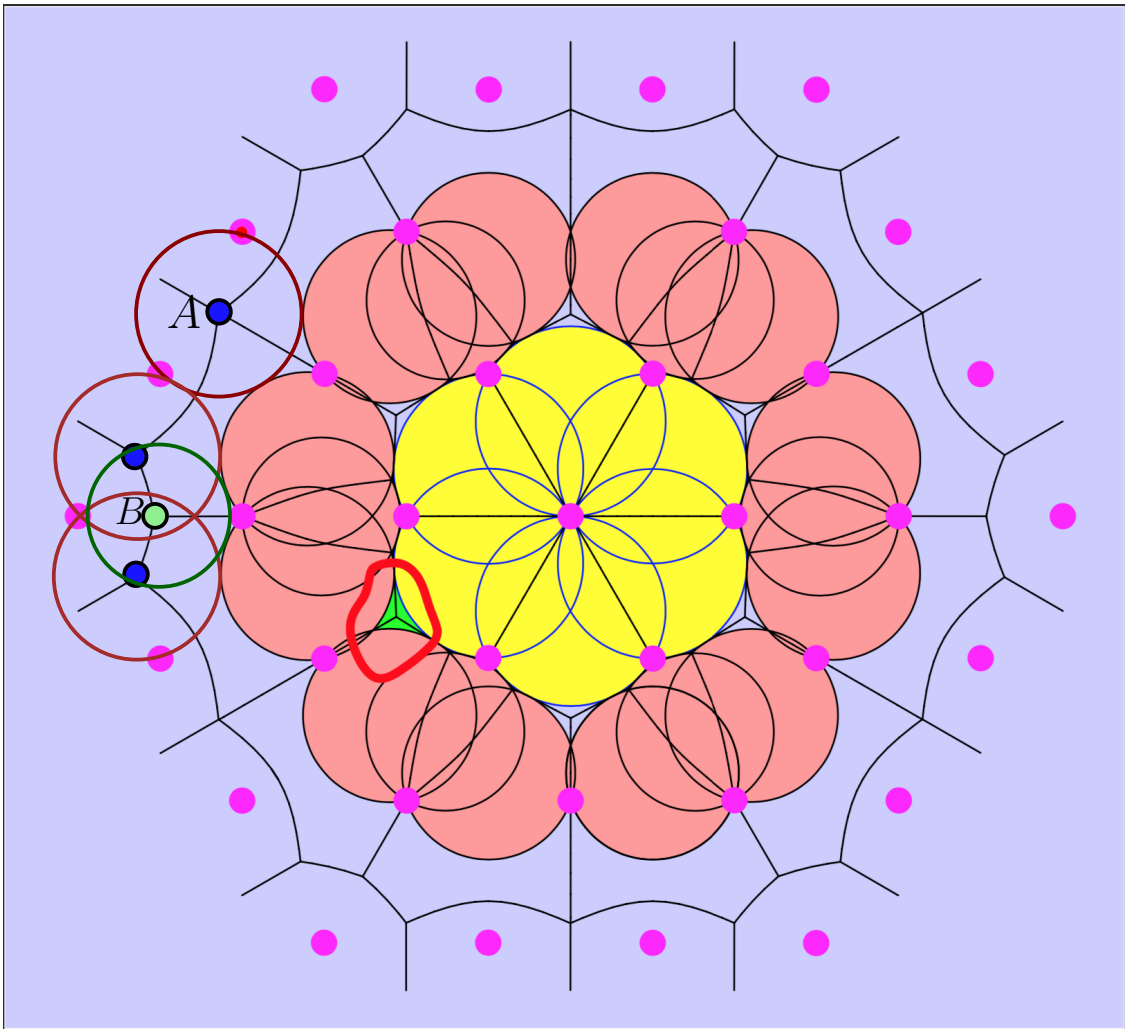


Figure 2.4: Two generations of disks inserted, and the three kinds of sites that we may encounter as the algorithm progresses. a) Dark blue points like point "A" are Voronoi vertices that actually contribute to the boundary of U_i for some i. They are adjacent to at least two new sites of the original diagram. b) Light green points like point "B" are Voronoi vertices that do not contribute to the boundary, but are harmless to add. They are adjacent to only one new site of the original Voronoi diagram, and equidistant to two other circles inserted over the course of the algorithm. c) The light green region between three circles encircled in red is a "pocket". This pocket corresponds to a middle of a hyperbolic edge from previous generation, which was covered by the inserted disks.

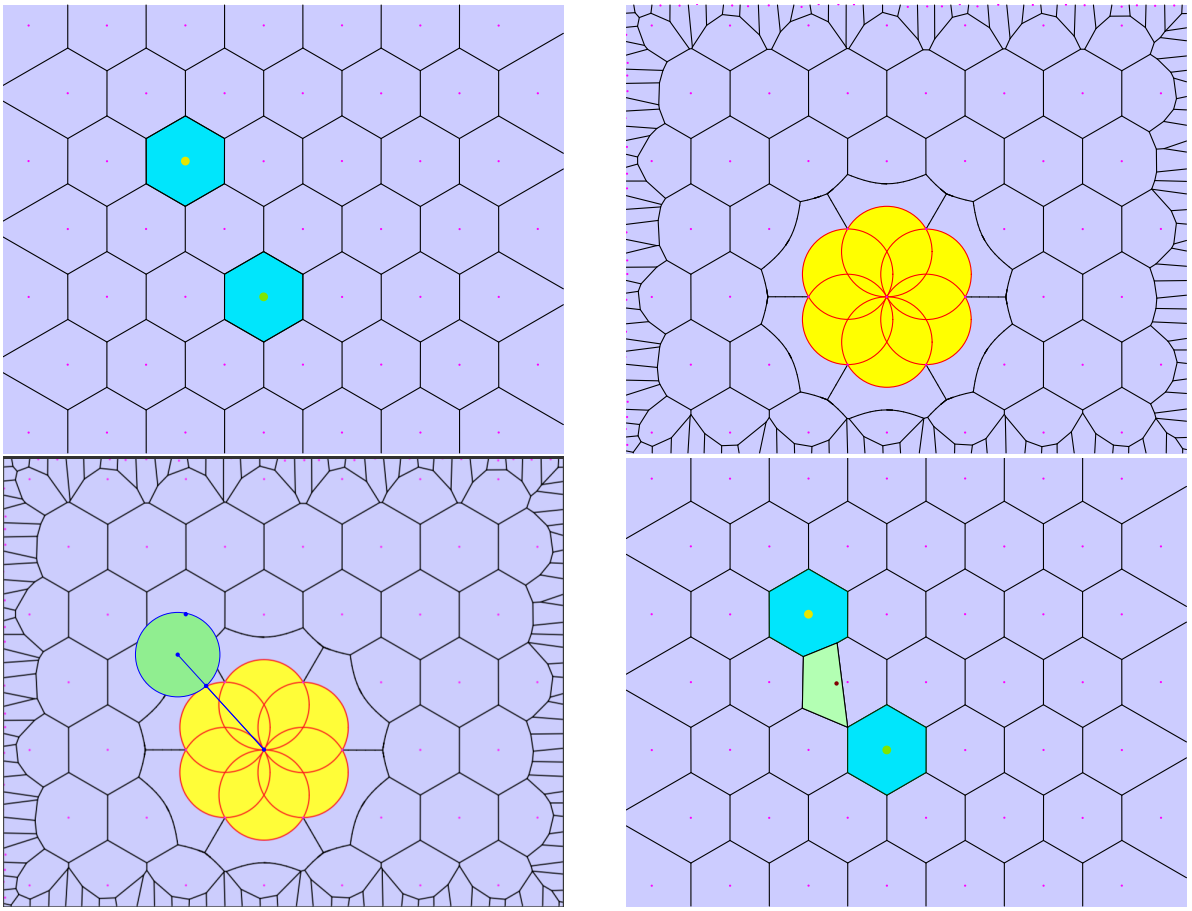


Figure 2.5: Reconstructing the shortest path

Reconstructing the shortest path. The basic reconstruction process is illustrated in Figure 2.5. In the figure, the sites to connect are marked in light blue. Then one iteration of the flowering process occurs. At this point, it is already the case that the region R_1 shares a boundary with the target cell. The reconstruction step involves first identifying the nearest vertex of the region R_1 to the target site. In the example there are two choices, and we arbitrarily pick one.

Now, the largest empty circle from this vertex touches the target site and touches (tangent to) one disk of U_1 at some point. This point of tangency is where we insert a point in the unweighted diagram.

This is justified as points on the boundary of U_1 will always just touch some vertex of R_0 by definition, and the point we chose will share the Voronoi vertex of the additive diagram with the target site's cell (as the target site is the closest site to the Voronoi vertex by construction), accomplishing the safe connection.

In general, the algorithm proceeds until some vertex of R_i is adjacent to the target site.

Then the largest empty circle at that vertex is (conceptually) created, and the point of tangency to U_i is computed. The point of tangency is inserted to the original diagram. By the discussion from earlier sections the cell of this point must touch the region R_{i-1} at some Voronoi vertex on its boundary. So we may continually repeat this process until we just touch some vertex of the region R_0, which is the Voronoi cell of the starting site.

In practice, computing the point of tangency is just computing the intersection of the line joining the centers of two Voronoi vertices with a disk. However to avoid the cost of nearest neighbor searches we remember for each type "A" Voronoi vertex the disk that generated it (during the insertion phase of the algorithm). Then the intersection and backtracking steps can be done in constant time per point of the optimal solution.

We note that the choice of points to be inserted is not unique. For instance, we could have chosen any point of the shared edges between R_i and the target cell to start the backtracking from. However it is convenient to only use vertices of the intermediate additive Voronoi diagram as the entities we would otherwise have to keep track of would be arcs rather than points.

2.3 RUNNING TIME ANALYSIS

A point p of the input is ***active*** at time i, if it is adjacent to a site inserted in the ith iteration.

Lemma 2.1. *An input point can be active for at most two iterations.*

Proof: An input point p is ***discovered*** at time i, if its Voronoi cell C in $\mathcal{V}_i = \mathcal{V}(\{U_i\} \cup (P \setminus U_i))$ is adjacent to the cell of U_i in this diagram.

Informally, once a cell is being discovered, in the next iteration of insertions, disks would inserted that touch it, and the following iteration it would be blocked from the (wave) front. Namely, the boundaries of C and R_i intersects, as R_i is the cell of U_i in $\mathcal{V}_i$. As such, p is on the boundary of U_{i+1}, or p might be contained in the interior of U_{i+1}. Indeed, consider a point q on the common boundary between C and R_i, and consider the disk of radius $\|p - q\|$ centered at q, and observe that this disk has p on its boundary, and it is contained in U_{i+1}, see Eq. (2.3). But that implies that C is contained in the Voronoi cell of U_{i+1} in $\mathcal{V}_{i+1} = \mathcal{V}(\{U_{i+1}\} \cup (P \setminus U_{i+1}))$. This cell is thus R_{i+1}. Thus, the cell of p might interact with cells created in U_i, and U_{i+1}. Clearly, p can not be adjacent to cells inserted in later iterations. QED

Let n_i be the number of the active input points at time i. Since inserting a disk, updating the Voronoi diagram, and discovering the new Voronoi sites is all done locally, and can be

charged to new entities created or deleted, the following is straightforward to verify. It is critical here that we are inserting sites centered at Voronoi vertices, which we already know their location – that is, there is no need to perform point-location query for the insertion. As such, we get the following.

Lemma 2.2. *The total running time of the ith iteration, is bounded by $O(\sum_{j=i-2}^{i+2} n_j)$.*

Proof: We only sketch the proof. The quantity stated above bounds the number of sites that the wavefront might interact with in the ith iteration. In particular, it bounds the number of sites inserted at time i. By Lemma 2.1, a site is active only for a constant number of iterations. A new site inserted which is adjacent only to inserted sites, is a pocket, and the algorithm does not insert it. As such, inserted sites must be adjacent to original input points. It is easy to verify that an input site can support only a constant number of such sites around it, which readily implies that while a site is active, only a constant number of new sites inserted might be charged to it.

This implies by planarity, that the total complexity of the Delaunay triangulation in the ith iteration is proportional the total complexity of the input points in the adjacent layers, which implies the claim. QED

Theorem 2.1. *The running time of the algorithm is $O(n \log n)$.*

Proof: Reading the input and computing the Voronoi diagram takes $O(n \log n)$ time. By Lemma 2.2, the total running time of the later stages is proportional to $\sum_i O(n_i) = O(n)$.QED

CHAPTER 3: SOME IMPLEMENTATION AND USAGE DETAILS

The algorithm was implemented in `C++` using CGAL [3]. Specifically, we use the 2D Apollonius Graphs implementation (which was implemented by Menelaos Karavelas and Mariette Yvinec), see [4, 5].

The source is available at bitbucket `https://bitbucket.org/vrdhrjn2/voronoi_insertion`. The repository also include an input file, and a script to run the program on various inputs.

The source code provides functionality to generate point sets and then to run the algorithm on them. At the end, the results of the algorithm are generated as four pdf files and one txt file.

The pdf files that are generated are:

1. A file of the initial Voronoi diagram with "start" and "end" cells highlighted for the user.

2. A file showing the BFS path along the Delaunay triangulation, for comparison with the optimal solution.

3. A file with an optimal sequence of points inserted into the unweighted diagram, realizing the safe connection of "start" and "end" cells.

4. A file showing the disks inserted, and the state of the underlying additive weighted Voronoi diagram at termination.

The point sets that are generated are of three types: diagrams of regular hexagonal Voronoi cells, diagrams of randomly chosen points, and diagrams of points representing the state of Illinois constructed from census data. The point sets are written into text files for the user to be able to reuse multiple times.

As discussed in earlier sections, bounding points along the edges of a bounding box of the point set are added for the underlying additive diagram phase of the algorithm, to ensure no edges are infinite. The start and end points are chosen by considering a rectangular subset of points in the interior and choosing the furthest two points among them. Then the algorithm is run until a path is found safely connecting the two cells.

Potential floating point or degeneracy issues are avoided by adding small perturbations to the point set as they are inserted into the initial diagram. Radii of the disks being inserted are also dilated by a very small factor for the same reasons.

We chose not to distinguish between type "A" and "B" points in the insertion phase of the algorithm, but use a generation variable to avoid pockets, as discussed earlier.

While we are not reporting the running times, they seem to be near linear, and agree with the theoretical analysis. To clarify, however, we mean that this is excluding the diagram construction phases that happen in the very beginning and at the very end of the code, when drawing the final results. The construction phases do have $O(n \log n)$ behavior, as expected for the construction of unweighted Voronoi diagrams.

We used the CGAL Apollonius Graph hierarchy for the implementation. The provided library insertion function has two phases - locating a nearest neighbor, and then constructing a new Voronoi cell by identifying conflict edges, etc. The nearest neighbor location in our algorithm can theoretically be done in constant time, because all the points we insert are Voronoi vertices whose adjacent sites are immediately known. However, we did not bypass the internal nearest neighbor search because the results of using the Apollonius Graph hierarchy were consistently linear time overall in practice.

CHAPTER 4: EXPERIMENTAL RESULTS AND DISCUSSION

We finally show in this section the types of point sets generated by our code and typical results of the application of our algorithm. We will discuss each of the hex, random, and Illinois point sets separately.

A summary of our experimental results are provided in Figure 4.22.

4.1 HEX MESHES

Throughout the exposition in this thesis we have relied on hexagonal meshes for typical illustrations of the basic idea of the algorithm.

Regular hexagonal meshes are particularly important in the context of this algorithm, because they provide a natural configuration for which our algorithm cannot do better than BFS on the dual graph. Indeed, it might seem that the results in Figure 4.22 disagree, but that is only because of the cumulative effect of the perturbations we use while reading the point set adding up to distort the graph just enough to allow our algorithm to create an improvement.

When the perturbations are not added in to the coordinates in the point set reading function, the number of points computed by BFS and our algorithm agree exactly, even on the 90 layer and 120 layer hexagonal meshes. Even with the perturbations, these large point sets only see improvements of about two points.

Point set generation To generate close to completely regular hexagonal Voronoi diagrams, we ask the user to input the number of points to uniformly space out in the horizontal direction. Then our code generates a roughly square region of roughly n^2 points with the right spacing in both dimensions for the grid to be hexagonal. Here is an example for a grid with $n = 30$, and with a "start" and "end" cell chosen for the purposes of the algorithm.

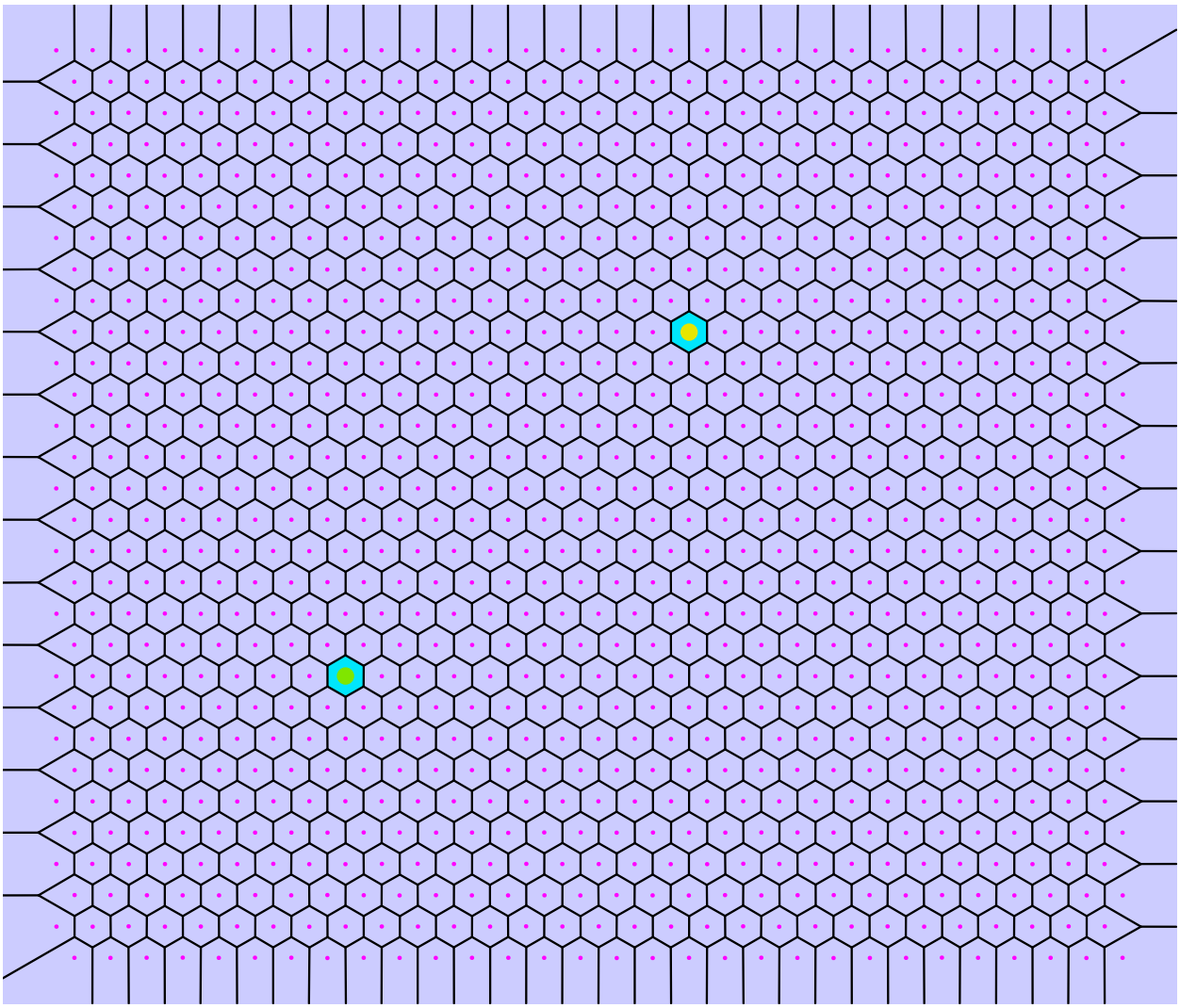


Figure 4.1: A 30 layer hexagonal grid produced by our code, with endpoints chosen. The start is at the lower left corner, in darker green. The end is on the upper right, in lighter yellow. This is the typical convention chosen for point sets except the Illinois points, where the start and end locations are inverted, i.e., we start at the upper right and try to securely find a path to a destination on the lower left.

BFS structure BFS on hex meshes is very straightforward, and the layers are larger and larger hexagonal regions of cells in contact with each other. A path chosen by BFS to link the cells from our previous cells is displayed here.

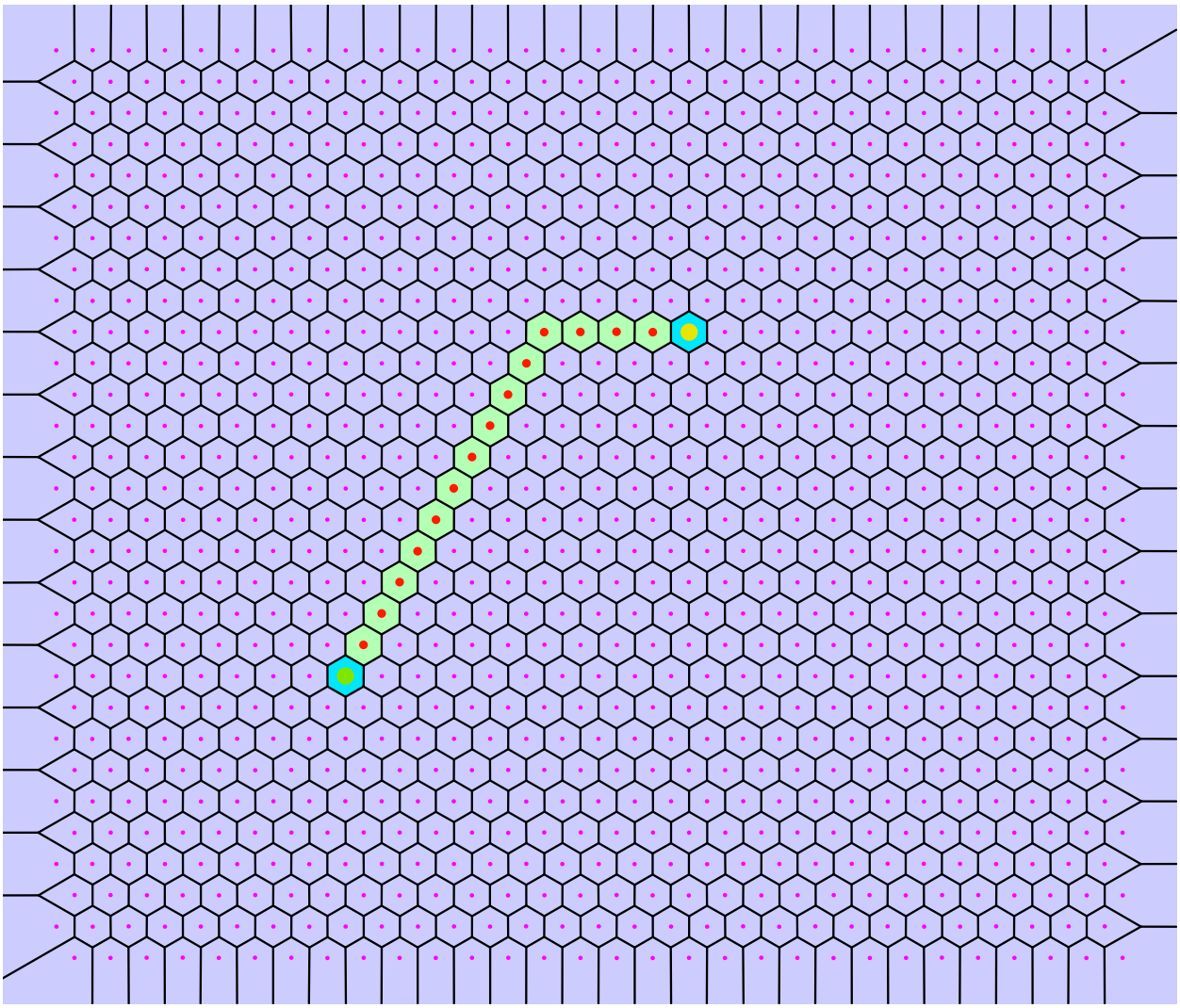


Figure 4.2: A BFS path produced by traversing the dual graph

Points chosen by the algorithm We expect the algorithm to perform only as well as BFS in the hex mesh case in terms of the number of points used. However the actual path chosen looks quite different from BFS and weaves through cells irregularly. A thing to be noted here is that the hyperbolic region after i steps is always slightly bigger than the union of i BFS layers, but still contained within $i+1$ BFS layers, for no net improvement in terms of reaching points but for an improvement in area.

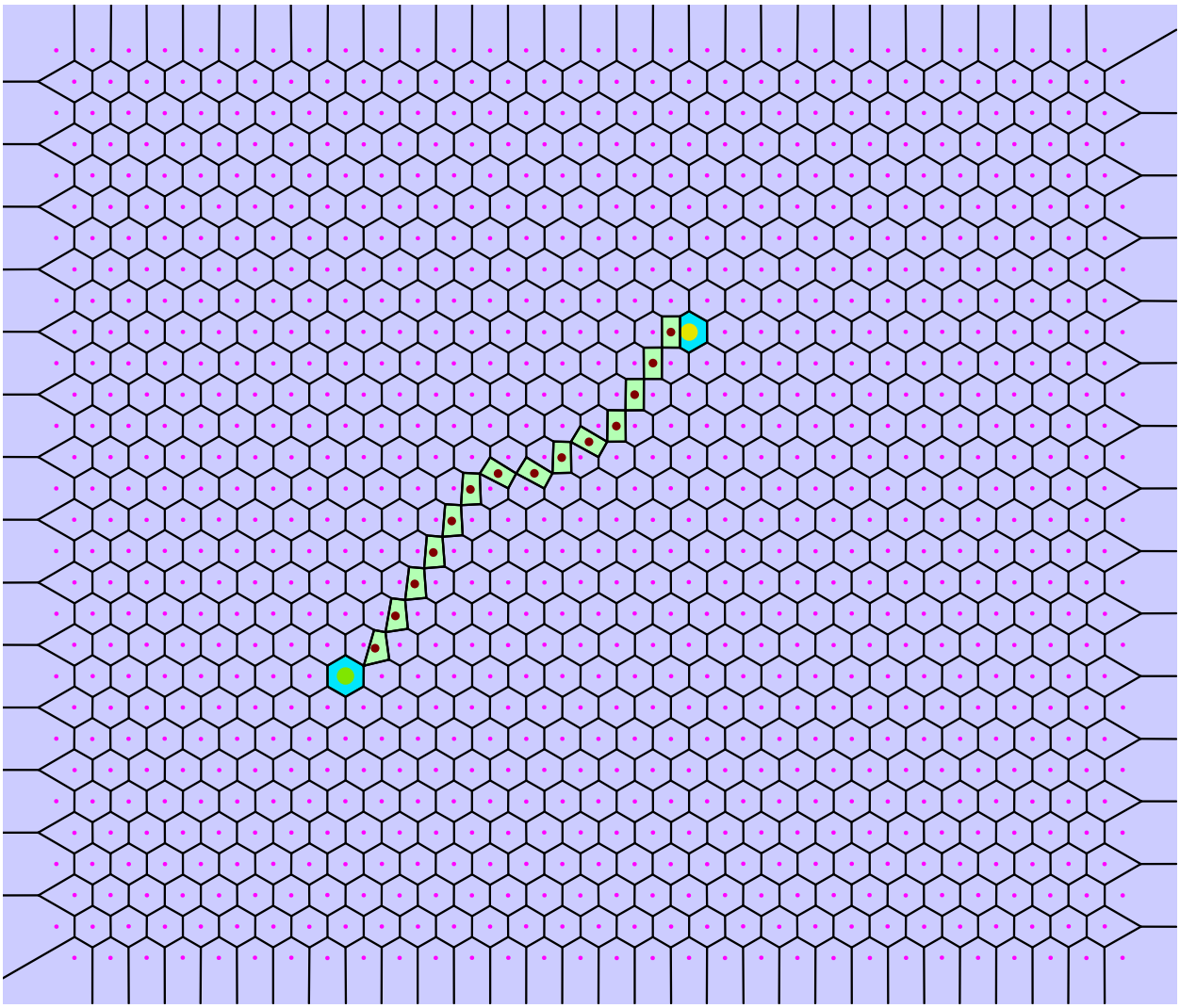


Figure 4.3: An optimal sequence of points computed by the algorithm

Structure of sites inserted The flowering process in the hex mesh case looks like a BFS traversal of hexagonally arranged disks. We have shown examples of this in miniature earlier in this thesis, but here is a full run as produced by our code, consisting of growing regions of disks and a hyperbolic wavefront ending at the target cell.

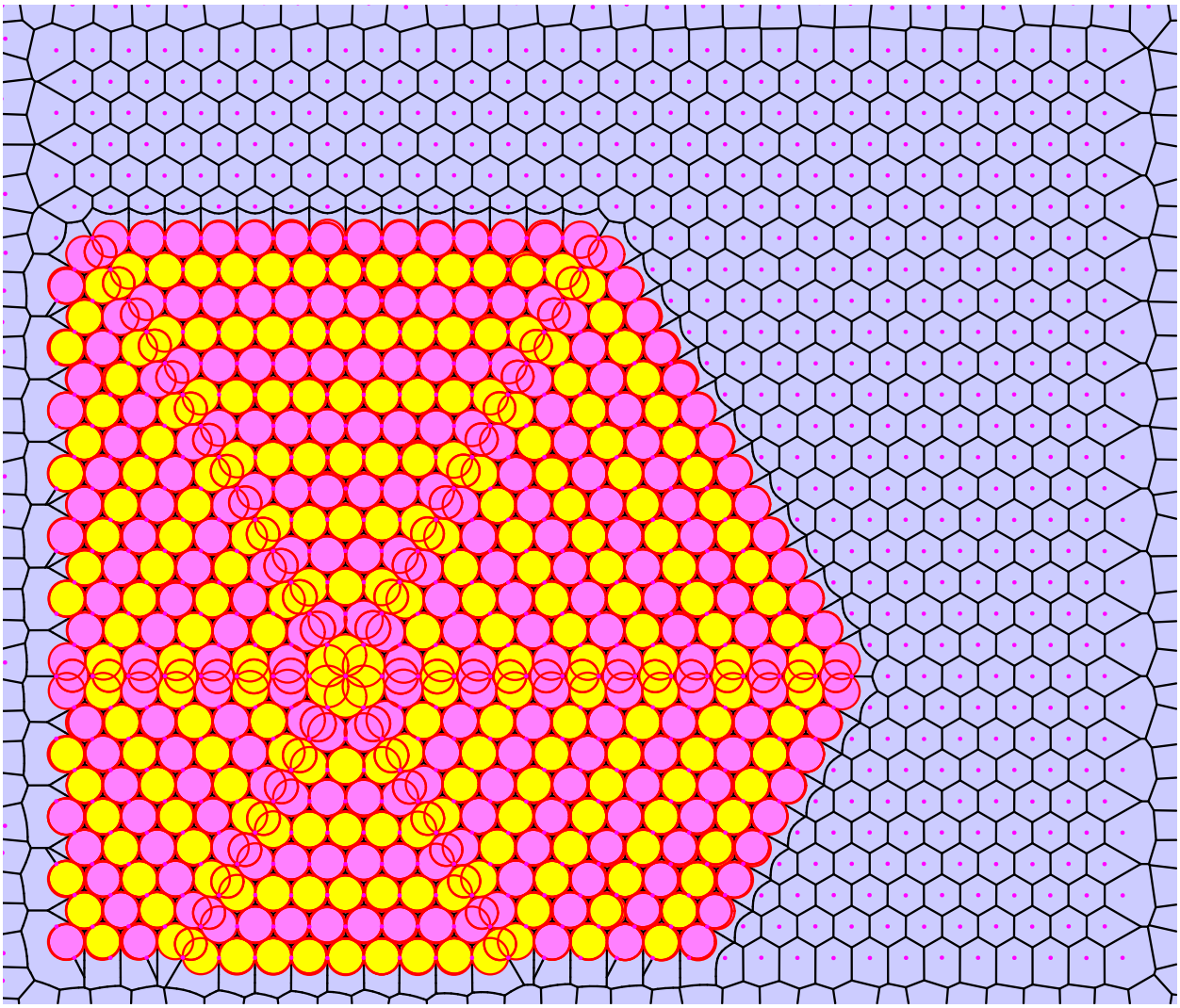


Figure 4.4: The disks inserted during execution. The reader is encouraged to zoom in to observe details such as the dark gaps actually being pockets of vertices correctly ignored by the algorithm (where three edges intersect), and the sites of the original diagram occurring exactly where circles of the same color touch each other, etc. Another interesting fact is the circles introduced by type "B" vertices themselves propagating in a hexagonal shape.

4.2 RANDOM MESHES

After the hex mesh experiments, random meshes are a welcome change because they almost always allow us to observe a significant improvement over BFS.

Point set generation Yet again, we generate points in a 2500x2500 unit region. However, this time the user directly specifies the number of points they would like and the mesh generation function would sample that many points in the square uniformly at random. For this section we will use the following random mesh of 2000 points, yet again with endpoints marked.

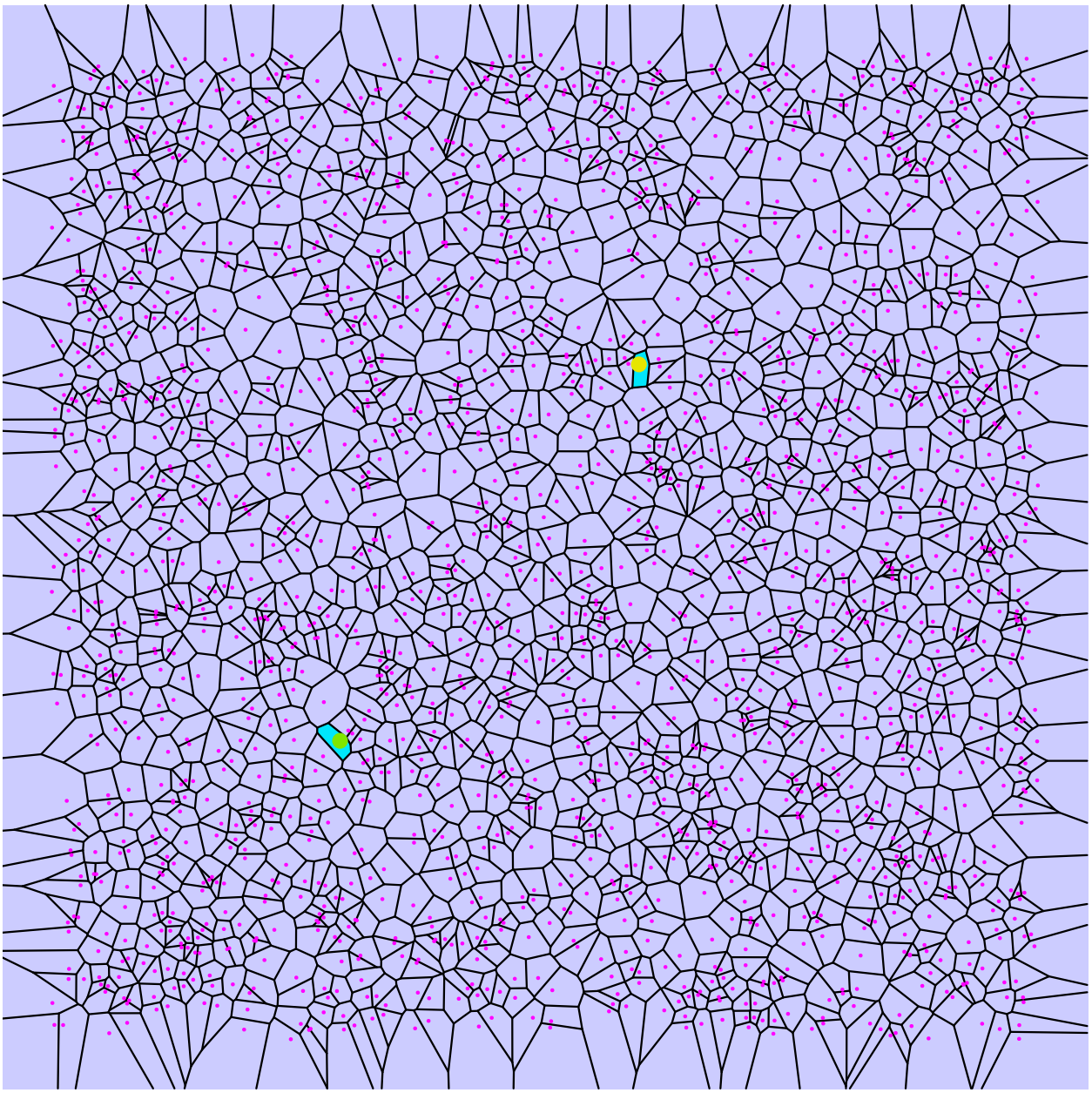


Figure 4.5: A random mesh of 2000 uniformly sampled points, with "start" and "end" cells marked

BFS structure Here is what a BFS path generated by our code connecting the two points in question looks like. The BFS solution of 16 intermediate points yet again looks quite good, but unlike the regular hex mesh case is nearly always going to be beaten by the algorithm.

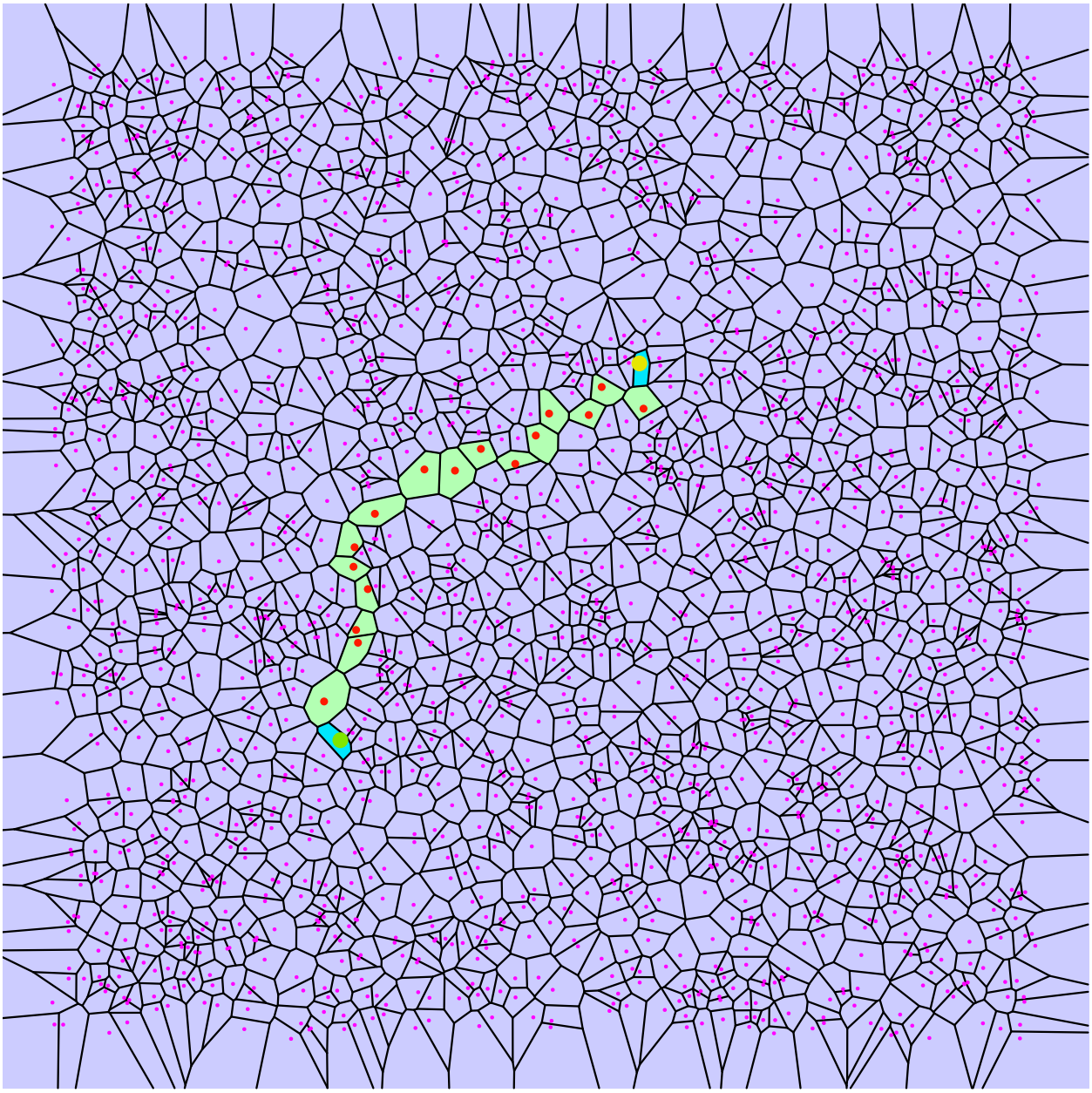


Figure 4.6: A BFS path of 16 intermediate points connecting the terminal cells

Points chosen by the algorithm Here are the points that our algorithm inserts to safely connect the cells. It beats BFS by a respectable 4 points with only 12 insertions.

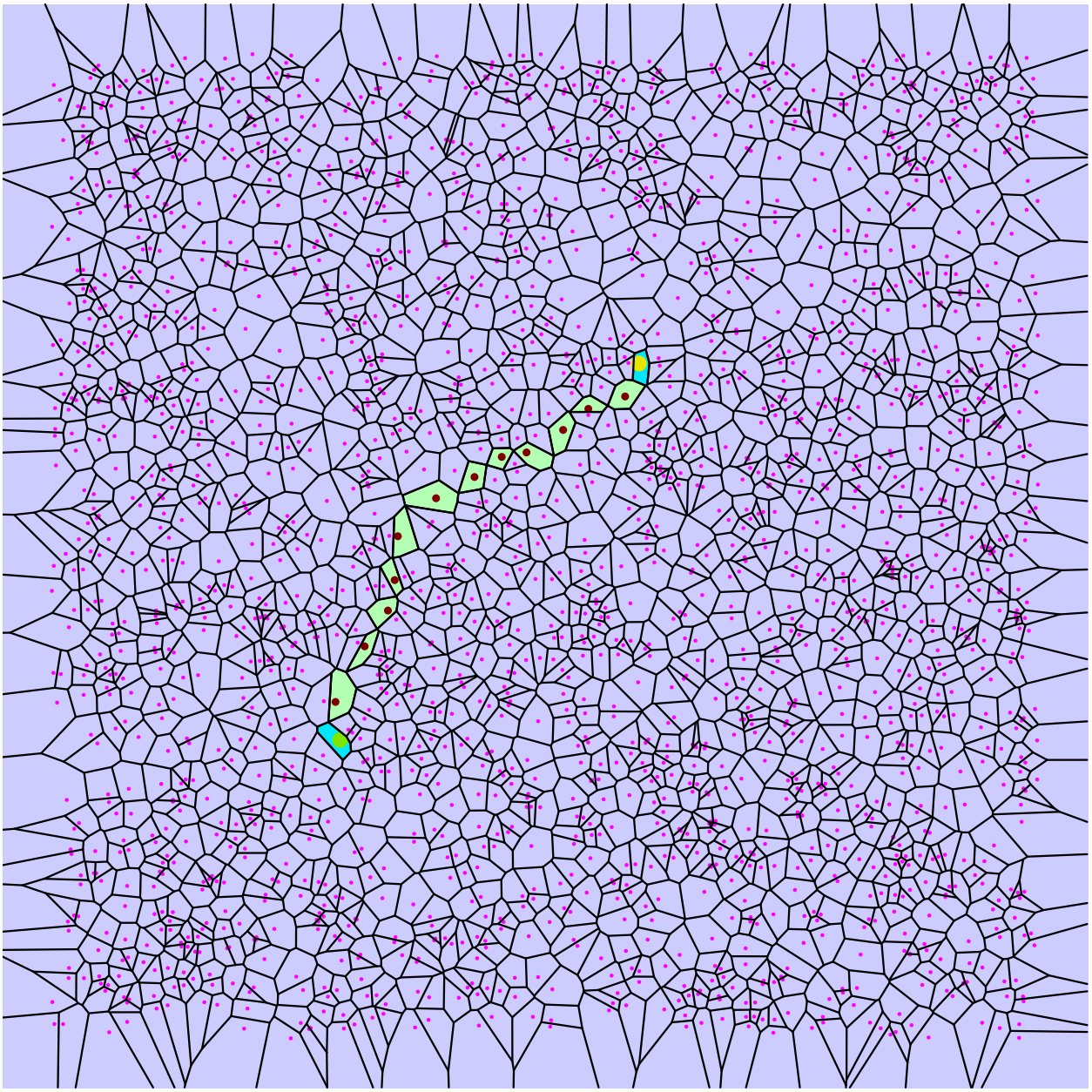


Figure 4.7: An optimal length sequence of 12 inserted points between the terminal cells

Disk structure Here is the diagram of disks inserted by the algorithm. Unlike the hex mesh case, it is quite irregular and has larger holes of many pockets clumped together in places.

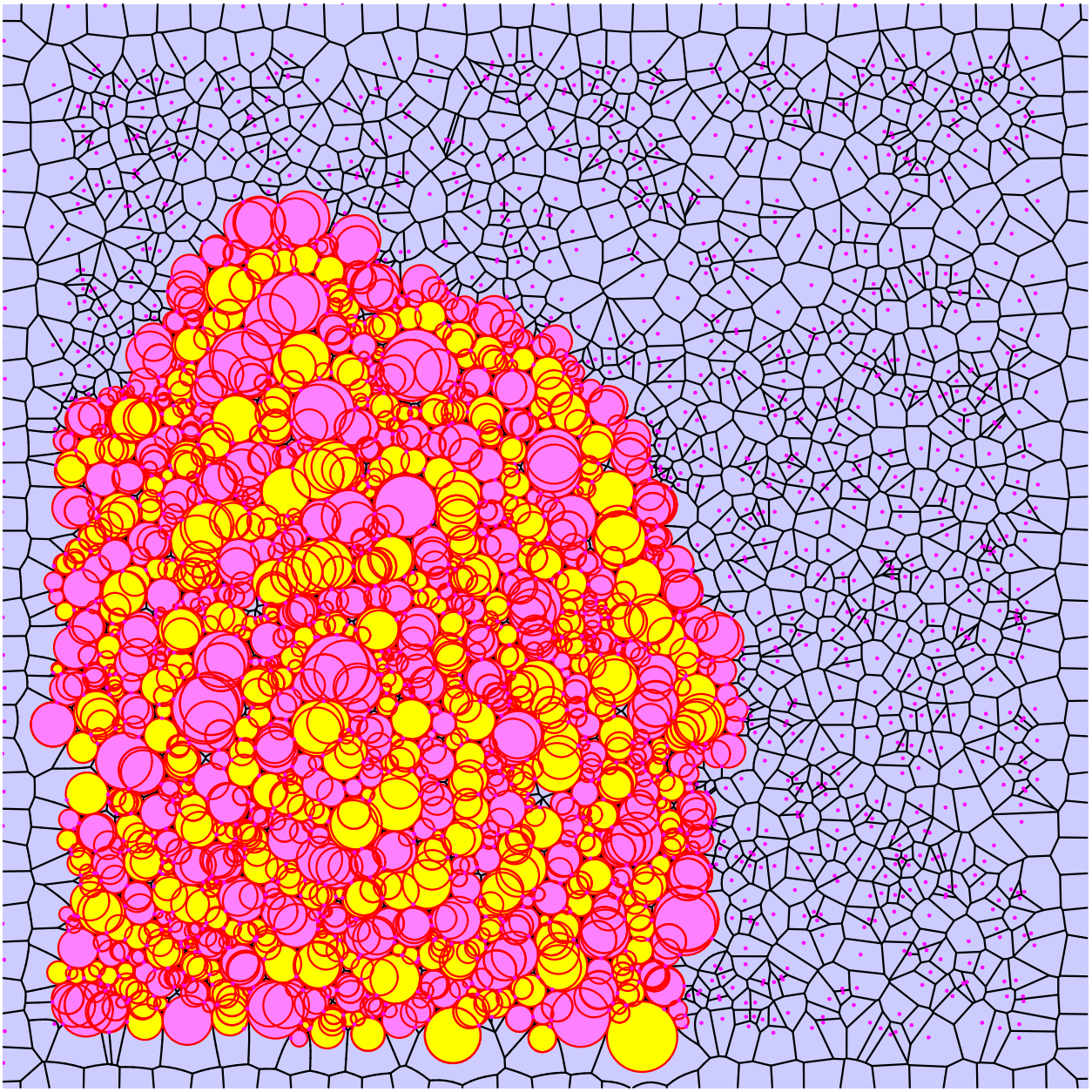


Figure 4.8: The disks inserted by the algorithm

4.3 ILLINOIS POINT SET

Finally, we turn our attention to the Illinois point set. Data on cities in Illinois was obtained from the census, and coordinates were mapped into our 2500x2500 square. The point set has 35252 points in it.

Sparsification The way we went about introducing different ways to use this point set was by sparsifying it. Our sparsification function independently either keeps or tosses a point with equal likelihood, and each successive sparsification roughly reduces points by half. Figure 4.10 shows us what the point set sparsified looks like over successive iterations.

Algorithm execution example First we demonstrate, as we did in the previous two cases, a typical example of running the algorithm on our Illinois point set. For an example of the algorithm behavior on it, let us consider an initial diagram sparsified five times, with "start" and "end" cells marked

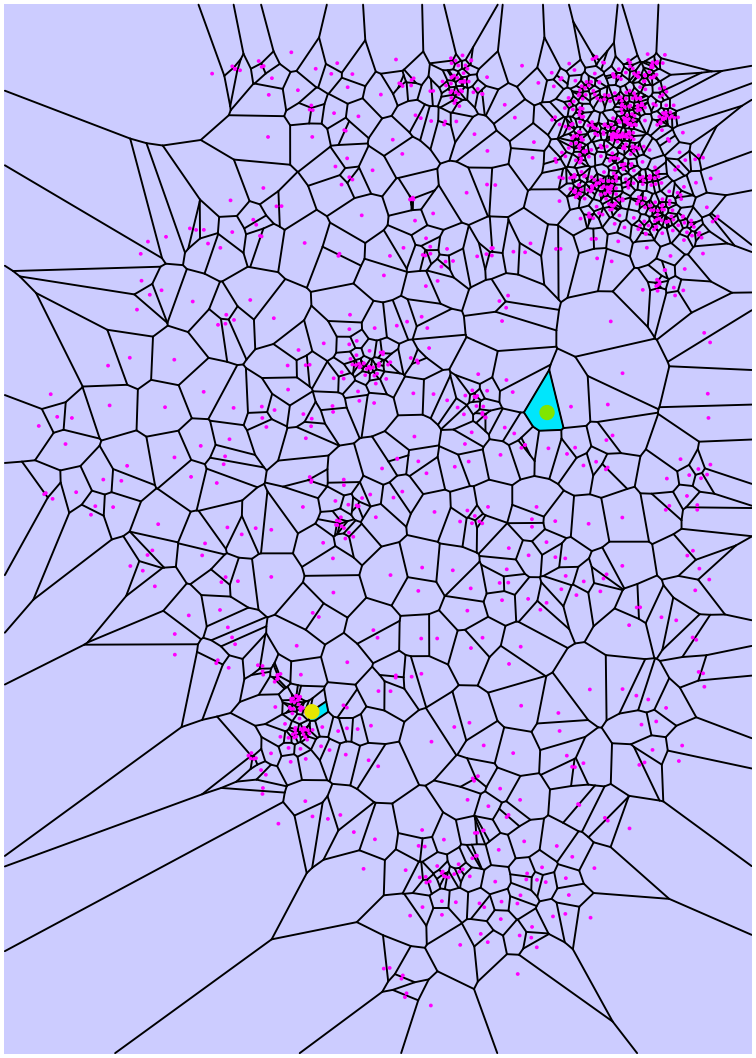
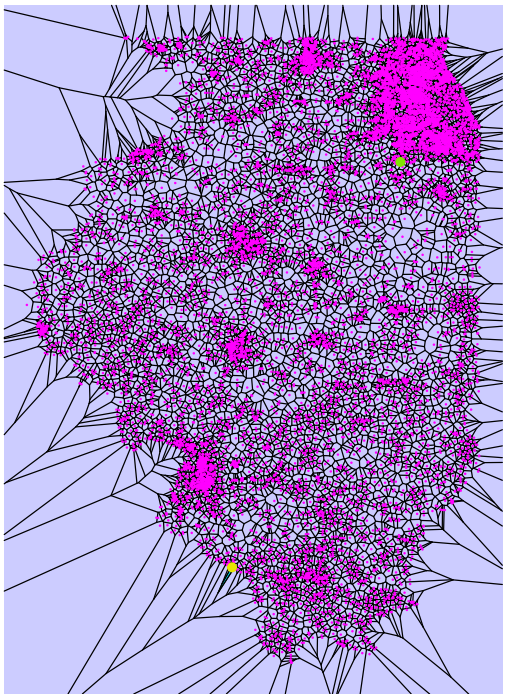
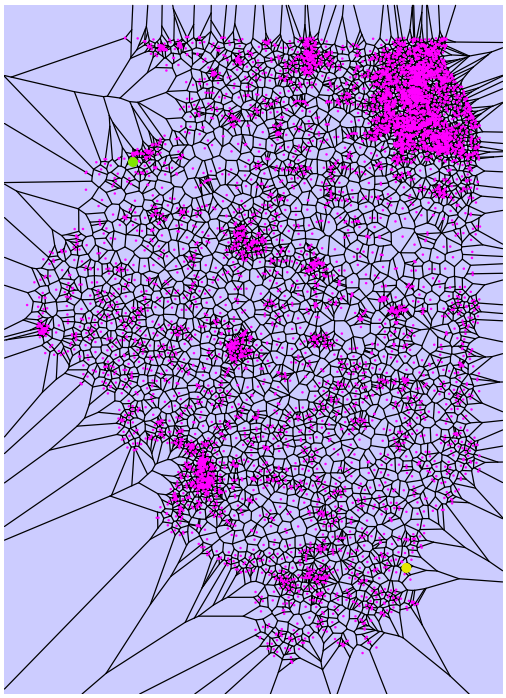


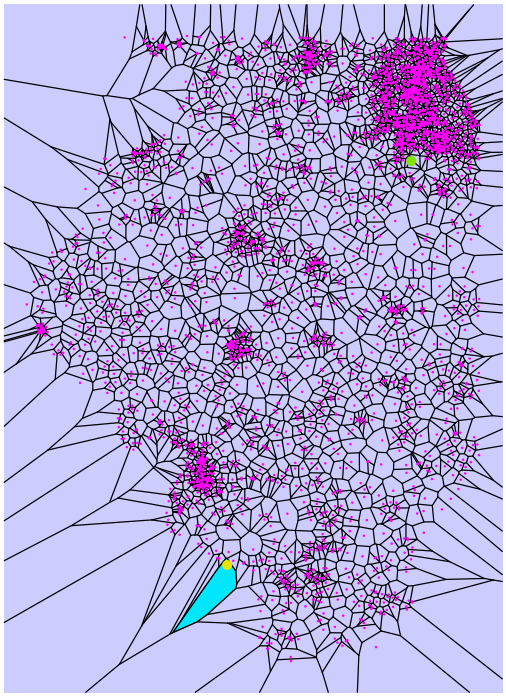
Figure 4.9: A five times sparsified illinois point set



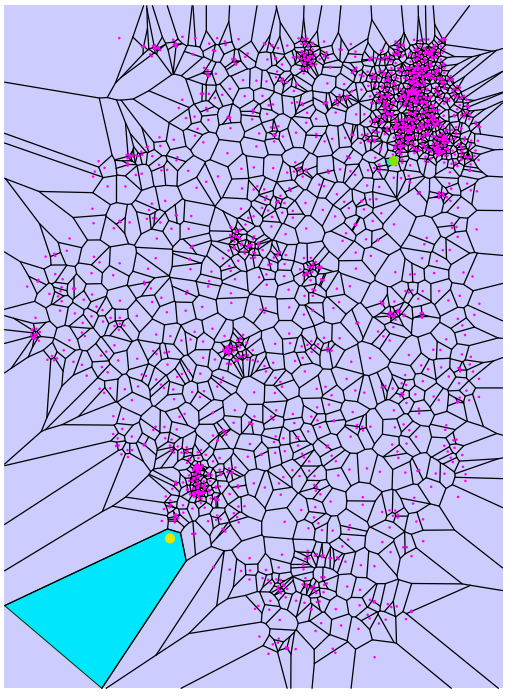
Sparsified once



Sparsified twice



Sparsified thrice



Sparsified four times

Figure 4.10: A point set made out of locations in Illinois (downloaded from the census and sparsified)

BFS distance for the 5 times sparsified set The BFS distance for our small Illinois example turns out to be 9 between the "start" and "end" cells

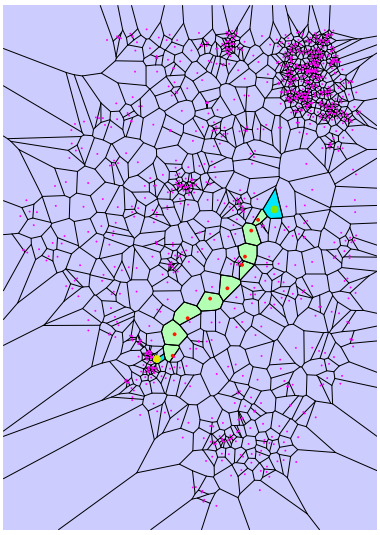


Figure 4.11: The BFS distance between the terminal sites

Optimal algorithm distance for the 5 times sparsified set Our algorithm manages to save one point over BFS and connect the cells in 8 insertions. However the diagram is already quite sparse so a meager improvement is not too surprising.

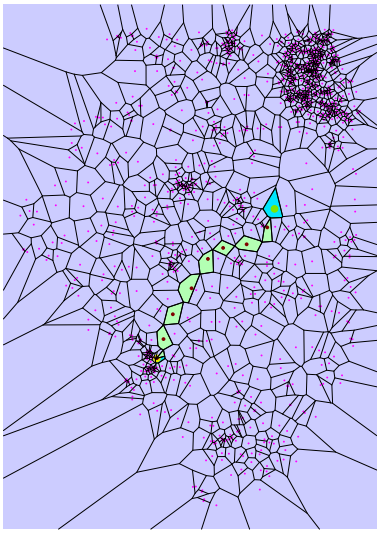


Figure 4.12: The optimal sequence of insertions to connect the sites

BFS distance for the 4 times sparsified set Just for comparison, we include the results of the algorithm on the four times sparsified set. Naturally one would expect the distance

to increase with the density of the set, and that is broadly true. For the choice of end points in the 4 times sparsified set, BFS finds a path of length 23 between the end points. However note how this path now leaves the implicit boundary of the point set, because of the irregularly concave distribution of points around the boundary of the Illinois point set.

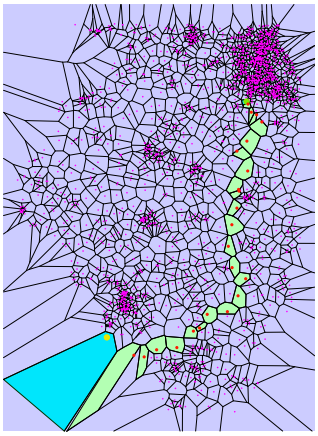


Figure 4.13: The BFS distance between the terminal sites

Optimal distance for the 4 times sparsified set The concave portions of the boundary also feature in this solution, which uses 17 points. The concavity allows points inserted outside the point set to still never meet the bounding vertices.

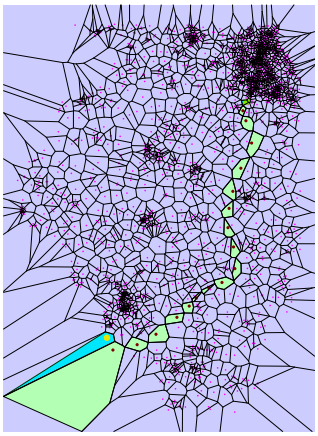


Figure 4.14: The optimal distance between the terminal sites

Disks for the 4 and 5 times sparsified sets For completeness we include what the inserted disks look like for the two sets we described earlier

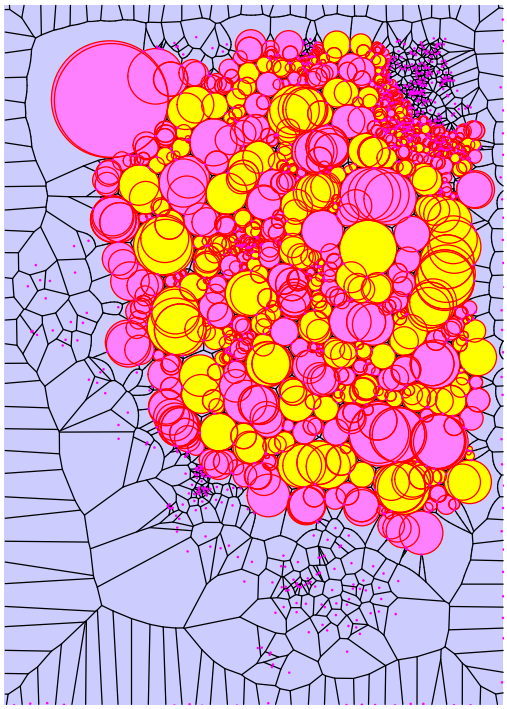


Figure 4.15: Disks inserted in the 5 times sparsified set

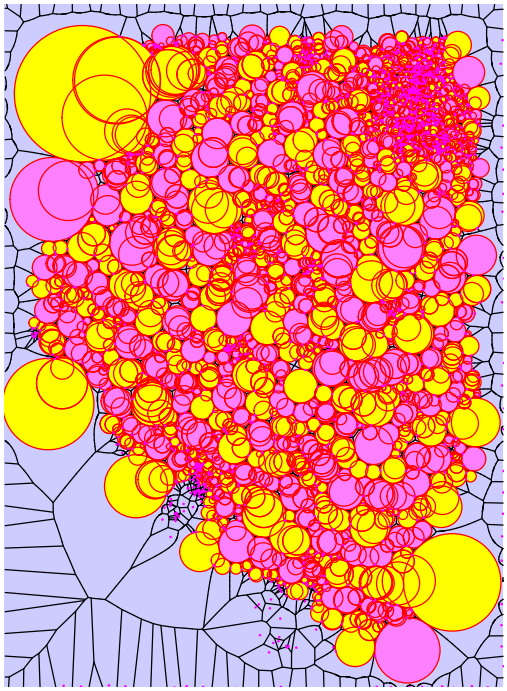


Figure 4.16: Disks inserted in the 4 times sparsified set

4.3.1 The once sparsified Illinois point set

We end by discussing an unusual way in which our algorithm behaved in experiments. From the previous two disk diagrams, it looks like the diagram sometimes chooses circles that skirt around the outside of the point set when the point set has parts that cave inwards on the boundary. Our implementation of both BFS and the algorithm stop propagating when a bounding point is reached. However, while that stops BFS from using any paths outside the input points' diagram, the algorithm manages to find ways to avoid the bounding points while staying outside the diagram to a large extent. The only way to stop the algorithm from doing so is to keep track of which region is the inside or outside on either side of the line joining edges on the boundary of the input points. This feature has not been added to our implementation and it is worthwhile to know that if one is actually trying to determine points at which to insert vertices for a real world purpose, the boundary or some kind of boundary function need to be readily specified (or one adds code to identify points on the edge and learn where the inside and outside are with respect to them).

The initial diagram is already available in Figure 4.10. Here is what the BFS solution joining the "start" and "end" cells looks like, and then we will see the algorithm's solution

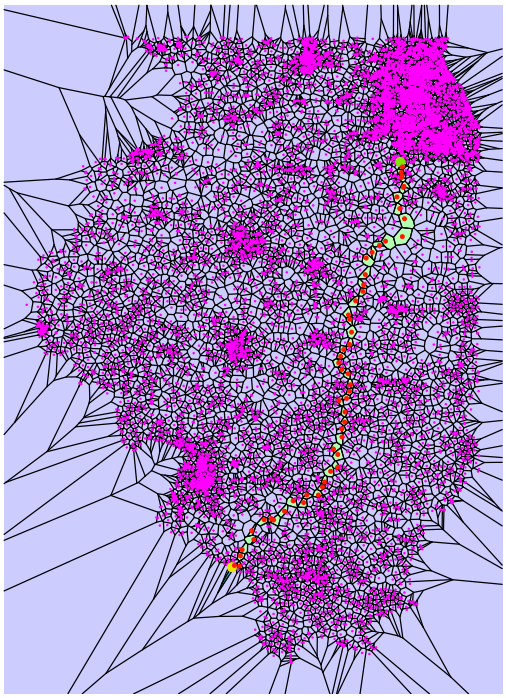


Figure 4.17: The 55-point BFS path for the once sparsified Illinois point set

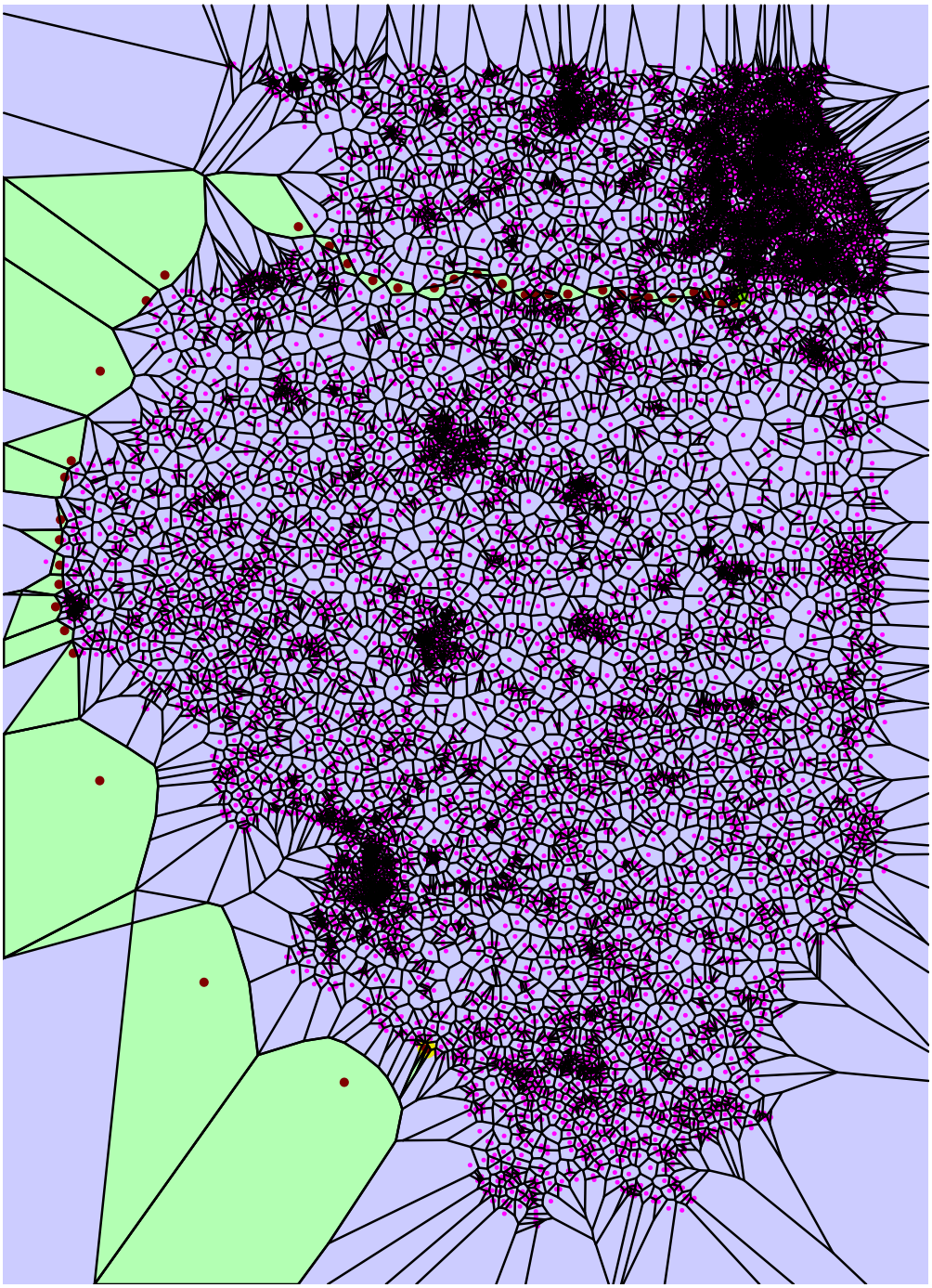


Figure 4.18: The 38-point optimal path for the once sparsified Illinois point set. The inserted points that are "outside" what we understand to be the interior of the input points actually form cells that do not touch the bounding vertices inserted. This persists even if we move the bounding vertices very close to the input, because of the concave portions of the boundary of the input Voronoi diagram. This situation can only be avoided by internally computing the boundary in some way, which we do not implement in the demos.

The disk picture helps understand how this sequence of insertions is valid, too:

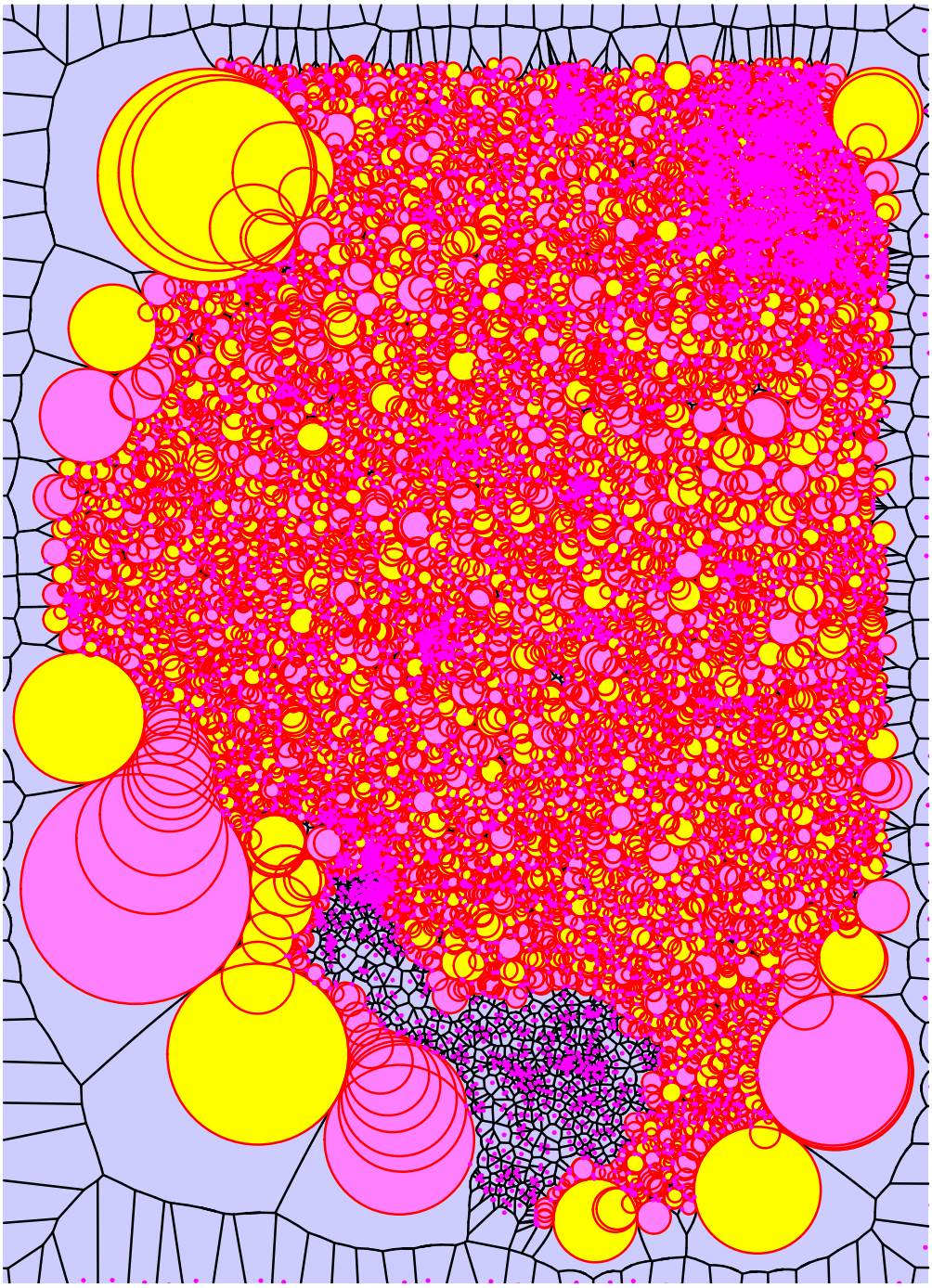


Figure 4.19: The disks inserted by the algorithm for the once sparsified Illinois point set. Many disks outside the point set are still shielded from the bounding points.

However, the algorithm is still useful on large point sets when choosing endpoints just a little away from the edges to avoid the phenomena observed before. Here is a comparison of

BFS and the algorithm on the same point set when the endpoints are moved inwards:

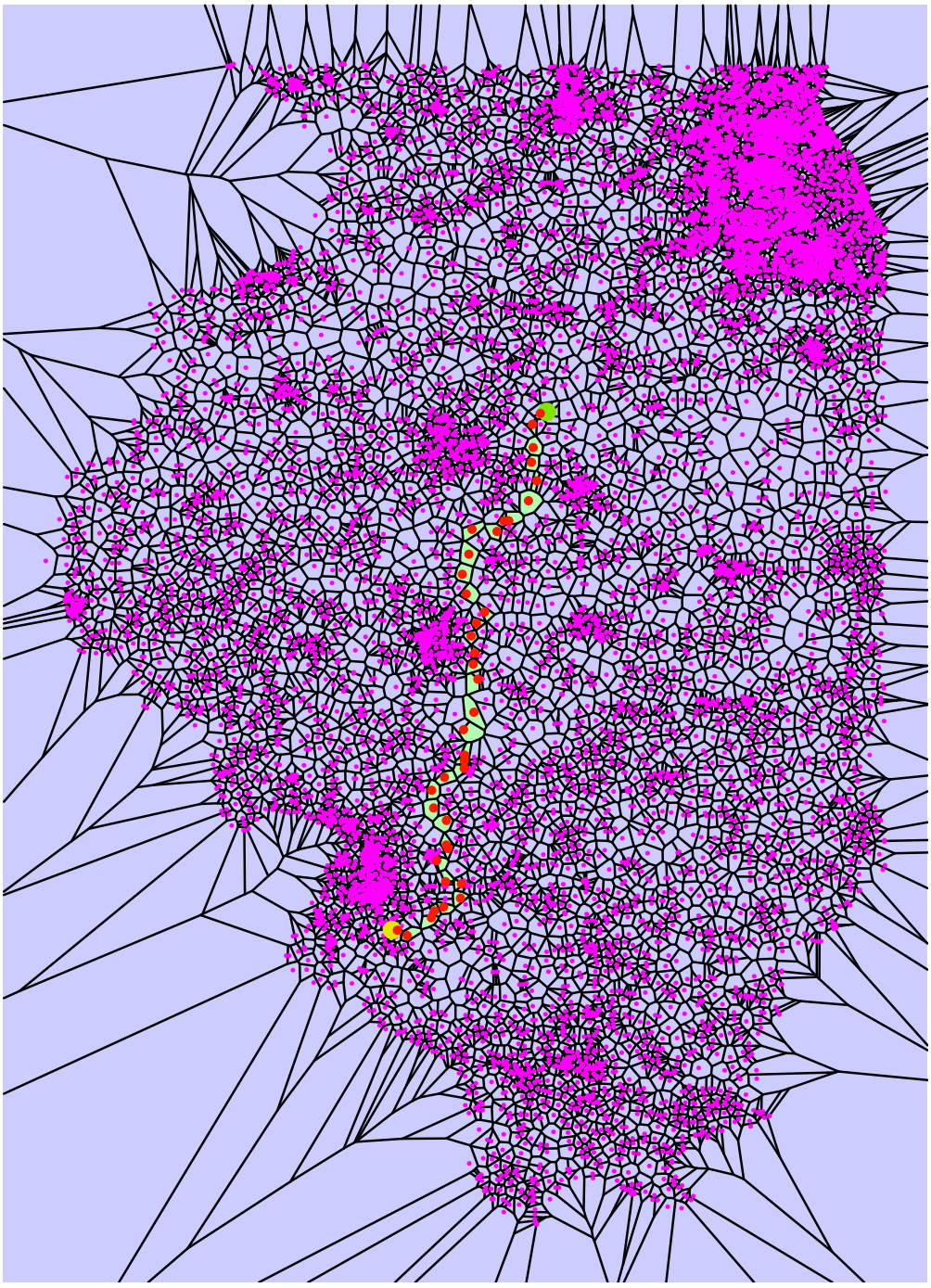


Figure 4.20: A 39-point bfs path on the once sparsified Illinois point set, with endpoints moved inwards

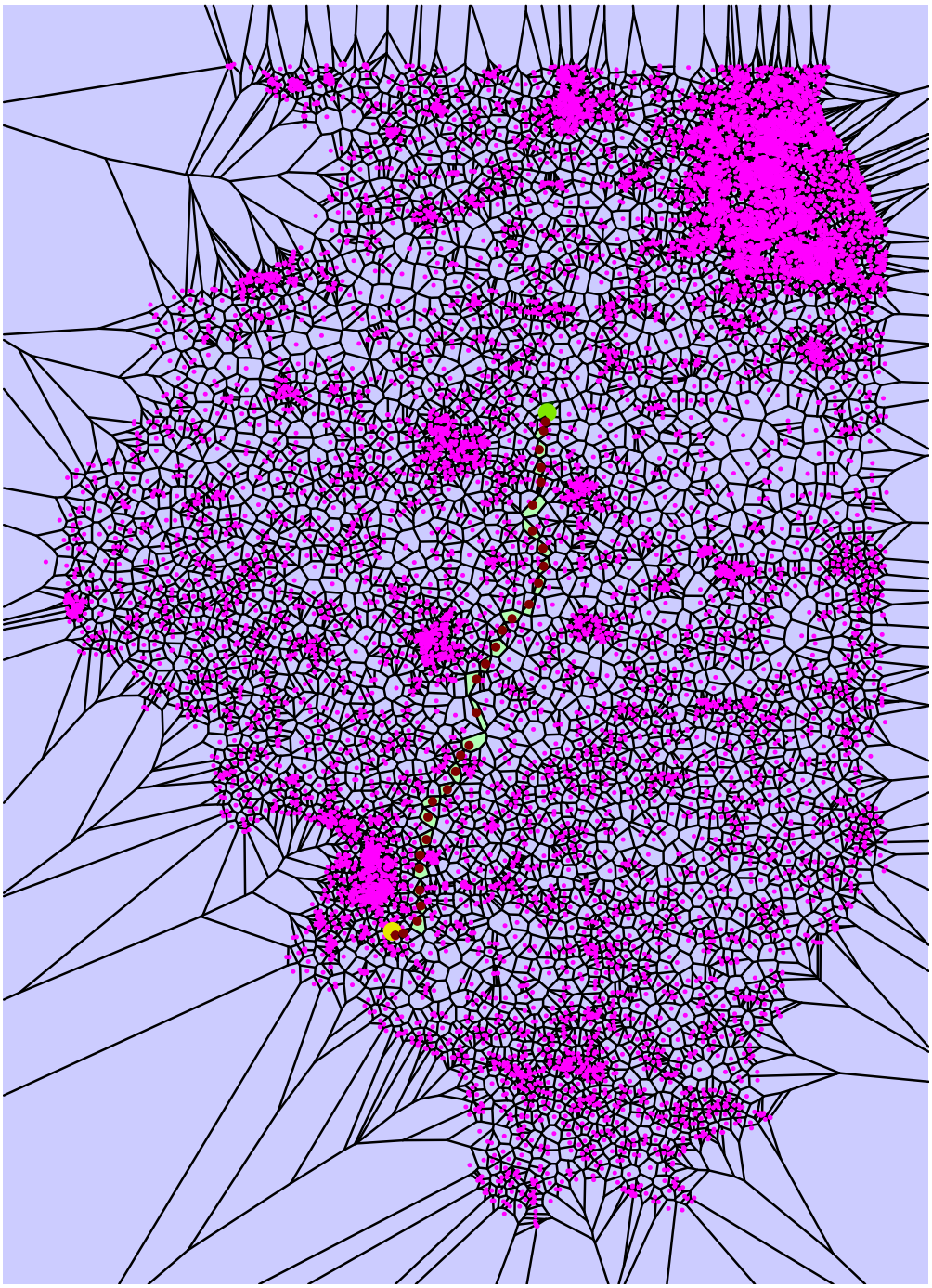


Figure 4.21: A 31-point optimal path that lies completely within the very large point set.

Input name	# points	BFS path	Algorithm path len
hex_010	100	5	5
hex_030	900	18	18
hex_060	3,600	56	54
hex_090	8,100	63	61
hex_120	14,400	77	75
i_d_02	17,560	55	38
i_d_04	8,816	39	28
i_d_08	4,435	30	23
i_d_16	2,242	22	15
i_d_32	1,123	17	12
i_d_64	559	11	9
rand_00100	100	5	4
rand_00200	200	8	6
rand_00400	400	12	9
rand_01000	1,000	17	14
rand_02000	2,000	28	21
rand_04000	4,000	37	30
rand_08000	8,000	47	38
rand_16000	16,000	76	61
rand_32000	32,000	95	74
rand_64000	64,000	137	110
rand_128000	128,000	206	164

Figure 4.22: Input used, and the algorithm performance on these inputs.

REFERENCES

[1] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," *SIAM J. Comput.*, vol. 16, no. 4, pp. 647–668, 1987. [Online]. Available: https://doi.org/10.1137/0216045

[2] J. Ruppert, "A Delaunay refinement algorithm for quality 2-dimensional mesh generation," *J. Algorithms*, vol. 18, no. 3, pp. 548–585, 1995. [Online]. Available: https://doi.org/10.1006/jagm.1995.1021

[3] The CGAL Project, *CGAL User and Reference Manual*. CGAL Editorial Board, 2020, 5.0.2. [Online]. Available: https://doc.cgal.org/5.0.2/Manual/packages.html

[4] M. Karavelas, "2D voronoi diagram adaptor," in *CGAL User and Reference Manual*. CGAL Editorial Board, 2020, 5.0.2. [Online]. Available: https://doc.cgal.org/5.0.2/Manual/packages.html#PkgVoronoiDiagram2

[5] M. Karavelas and M. Yvinec, "2D apollonius graphs (delaunay graphs of disks)," in *CGAL User and Reference Manual*. CGAL Editorial Board, 2020, 5.0.2. [Online]. Available: https://doc.cgal.org/5.0.2/Manual/packages.html#PkgApolloniusGraph2