Construction d'un diagramme de Voronoï

Aël De La Rosa–Leridon, 42314

4 juillet 2025

Résumé

De manière informelle, un diagramme de Voronoï d'un ensemble de points d'un espace de dimension d, appelés germes est une partition de l'espace en « zones d'influence » de ces germes, c'est-à-dire, pour chacune d'entre elles l'ensemble des points qui sont plus proches d'un germe donné que de tous les autres.

Nous allons ici nous intéresser au cas particulier de la dimension 2, en définissant la « zone d'influence » d'un germe comme les points du plan plus proches de celui-ci que de tous les autres pour la distance euclidienne.

L'objectif de ce projet est l'implémentation de l'algorithme de Fortune, permettant de calculer le diagramme de Voronoï associé à n points du plan en temps $O(n \ln n)$

1 Définitions

Soient $n \in \mathbb{N}^*$, $P = \{p_1, ..., p_n\} \subset \mathbb{R}^2$.

Définition 1.1. On appelle *cellule de Voronoï* associée au point p_i l'ensemble

$$V(p_i) = \{p \in \mathbb{R}^2 / \ \forall j \neq i, \quad \|p_j - p\| \geq \|p_i - p\|\}$$

Définition 1.2. Le *diagramme de Voronoï* associé à l'ensemble des germes P est :

$$D(P) = \bigsqcup_{i=0}^{n} V(p_i)$$

Définition 1.3. L'*arête de Voronoï* entre deux points p_i et p_j est définie comme :

$$a(p_i, p_j) = \{p \in \mathbb{R}^2 / \ \|p - p_j\| = \|p - p_i\| \text{ et } \forall k \notin \{i, j\}, \quad \|p - p_k\| \geq \|p - p_i\|\}$$

2 Représentation du diagramme

2.1 Structures de données

La structure de données généralement utilisée pour représenter un diagramme de Voronoï est une *DCEL* (Doubly Connected Edge List). Une arête $a(p_i, p_j)$ sera alors représentée par deux *demi-arêtes* jumelles, orientées dans des sens contraires (cf. Figure 2). Une cellule sera ainsi formée d'une liste circulaire doublement chaînée de demi-arêtes, décrivant les arêtes de la cellule dans le sens trigonométrique.

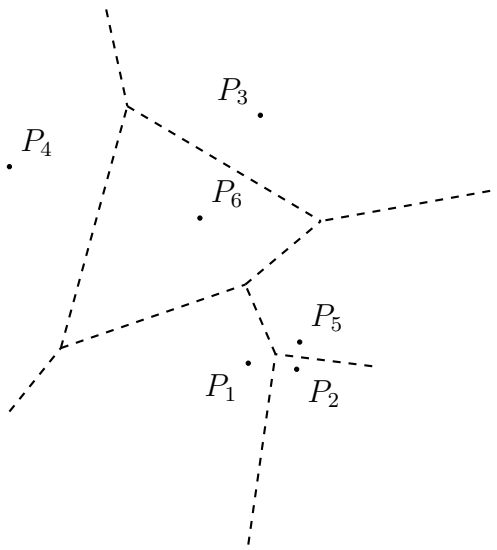


FIGURE 1 – Un Diagramme de Voronoï

```
1 struct demi_arete {
2     int ind;
3     struct demi_arete *jumelle;
4     pointf debut;
5     struct demi_arete *prev;
6     struct demi_arete *next;
7 };
```

FIGURE 3 – Implémentation en C

2.2 Complexité spatiale

Lemme 2.1. Pour $n \geq 3$, le nombre de sommets n_s du diagramme vérifie $n_s \leq 2n-5$ et son nombre d'arêtes n_a est tel que $n_a \leq 3n-6$

Preuve. Quitte à ajouter un sommet à l'infini, le diagramme de Voronoï peut être interprété comme un graphe planaire. Notons S l'ensemble des sommets de ce graphe, et A l'ensemble de ses arêtes. La formule d'Euler sur les polyèdres donne alors

$$|S| - |A| + n = 2$$

Les cellules de Voronoï étant convexes, on a de plus

$$\forall s \in S, \quad \deg s \geq 3$$

Il vient alors

$$2|A| = \sum_{s \in S} \deg s \geq 3|S|$$

On obtient ainsi

$$\begin{cases} |S| \leq 2n-4 \\ |A| \leq 3n-6 \end{cases}$$

Or par construction

$$|S| = n_s + 1 \text{ et } |A| = n_a$$

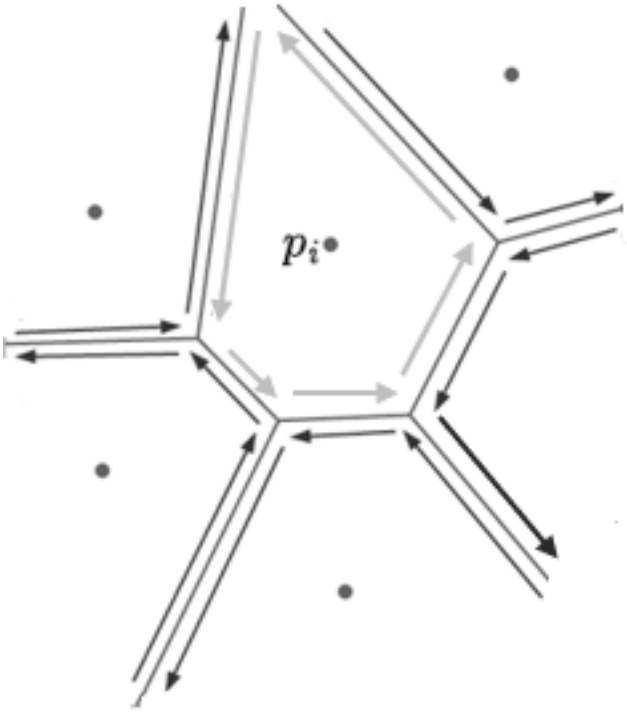


FIGURE 2 – Représentation des cellules avec une DCEL

D'où le résultat.

On aboutit alors au résultat suivant :

Théorème 2.1. La complexité spatiale du diagramme de Voronoï associé à n points est en $O(n)$

3 Algorithme de Fortune

3.1 Principe général

Si des algorithmes utilisent une approche incrémentale pour construire un diagramme de Voronoï, l'algorithme de Fortune se base sur le principe d'*algorithme à balayage* pour effectuer cette construction, cette approche ayant l'avantage que la partie du diagramme déjà construite à un instant de l'exécution n'aura pas à être modifiée par la suite.

3.1.1 Ligne de rive & ligne traversante

Dans l'algorithme de Fortune, le balayage du plan s'effectue à l'aide de deux lignes traversant le plan :

- La *ligne traversante*, ou *sweepline* Q est une droite horizontale traversant le plan de bas en haut et « découvre » divers *évènements* qui seront traités par l'algorithme.
- La *ligne de rive* ou *beach line* L est celle qui construit le diagramme : située en dessous de la ligne traversante, elle représente l'ensemble des points aussi proches d'un germe découvert dont la cellule n'est pas encore totalement construite que de la ligne de rive. La ligne de rive est ainsi formée d'arcs de parabole, dont les intersections (*breakpoints*)

tracent, au fur et à mesure que la ligne traversante avance, les arêtes du diagramme de Voronoï.

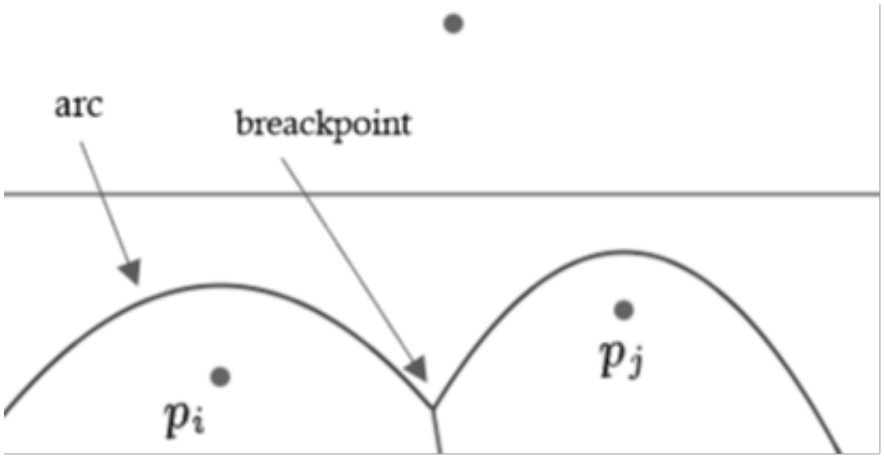


FIGURE 4 – Ligne de rive & ligne traversante

3.1.2 Circle & Site events

Deux types d'évènements sont détectés par la *sweepline* : les *circle events* et les *site events*.

— Un *site event* correspond à la découverte d'un germe. Leur emplacement est alors une donnée du problème.

— Un *circle event* correspond à la découverte d'un cercle passant par exactement 3 germes et tel qu'aucun germe ne lui est intérieur. Puisque le centre d'un tel cercle est à équidistance des 3 germes, il correspond à un sommet du diagramme. Les *circle events* sont alors calculés pendant l'exécution de l'algorithme, et susceptibles d'être des faux positifs, si jamais un germe est détecté par la suite à l'intérieur du cercle. Lorsqu'un *circle event* est effectivement confirmé, il y a de plus disparation d'un arc dans la ligne de rive.

3.2 Algorithme

Dans ce qui suit, si a est un arc de L, on note a_g l'arc à gauche de a et a_d l'arc à droite de a, et les opérations qui les font intervenir sont effectuées sous réserve de leur existence.

Par ailleurs, les détails des fonctions non spécifiées dans cette partie seront donnés en annexe.

Algorithme 1 : GèreCircleEvent

Entrée : *circle event* e confirmé, ligne de rive L, ligne travsersante Q, DCEL D
Résultat : Effet de bord

$a := e$.arc
$e_1 :=$ CircleEvent(a.point, a_d.point, a_{dd}.point)
$e_2 :=$ CircleEvent(a.point, a_g.point, a_{gg}.point)
$Q \leftarrow Q \setminus \{a_d$.evenement , a_g.evenement$\}$
a_d.evenement $\leftarrow e_1$
a_g.evenement $\leftarrow e_2$
MetAJourAretes(a.arete, a_g.arete)
AjouteAretesBis(a,D)
$L \leftarrow L \setminus \{a\}$

Algorithme 2 : GèreSiteEvent

Entrée : *Site event* e détecté, ligne traversante Q, ligne de rive L, DCEL D
Résultat : Effet de bord

$p := e$.site
si $L = \emptyset$ **alors**
    $L \leftarrow \{$Parabole générée par $p\}$
sinon
    Rechercher dans L l'arc a_0 au-dessous de p.
    $Q \leftarrow Q \setminus \{a_0$.evenement$\}$
    Scinder a_0 en a_1 et a_2 et insérer la parabole générée par p, entre a_1 et a_2
    $e_1 =:$ CircleEvent (a_g.site, a_1.site , p)
    $e_2 =:$ CircleEvent (p, a_2.site , a_d.site)
    a_1.evenement $\leftarrow e_1$
    a_2.evenement $\leftarrow e_2$
    AjouteAretes(D, a.site, p)
    $Q \leftarrow Q \sqcup \{e_1, e_2\}$

Algorithme 3 : Fortune's Algorithm

Entrée : L'ensemble des germes P
Résultat : La DCEL associée à $D(P)$

$D \leftarrow \emptyset$
$Q \leftarrow P$
$L \leftarrow \emptyset$
tant que $Q \neq \emptyset$ **faire**
    $e :=$ evenement d'ordonnée la plus basse de Q
    $Q \leftarrow Q \setminus \{e\}$
    si e.estCercle **alors**
        GèreCircleEvent(e, L, Q, D)
    si e.estSite **alors**
        GèreSiteEvent(e, L, Q, D)
retourner D

4 Implémentation

4.1 Les évènements et la ligne traversante

Le pseudocode de l'algorithme de Fortune nous incite à représenter Q à l'aide d'une file de priorité, implémentée à l'aide d'un *TasMin*, pour garantir une insertion en $O(\ln n)$ et un dépilement en $O(1)$. On ne retirera pas à proprement parler les *circle events* invalidés de Q mais on mettra simplement leur champ `.estCercle` à *false* pour signaler qu'ils n'ont plus à être traité.

La priorité d'un *site event* sera l'ordonnée du germe auquel il correspond, tandis que celle d'un *circle event* sera le point d'ordonnée maximale du cercle correspondant. Ainsi, lorsqu'un *circle event* est dépilé, et qu'il n'a pas été invalidé cela signifie bien qu'aucun point n'a été découvert à l'intérieur du cercle.

Il est important de noter qu'avec une telle implémentation, l'algorithme de Fortune n'est correct que sous certaines hypothèses sur P, sans lesquelles l'intégrité de L et de Q pourrait être compromise :

— 4 points de P ne sont jamais cocirculaires.
— Les abscisses des points de P sont deux à deux distinctes, de même que leurs ordonnées.
— Tous les points ne sont pas situés sur une même ligne.

```c
struct evenement {
    bool est_cercle;
    bool est_site;
    double prio;
    int ind;
    pointf p;
    chainon *arc;
};

struct ligne {
    int taille;
    int capacite;
    evenement **data;
};
```

Figure 5 – En C

4.2 Représentation de la ligne de rive

La ligne de rive est généralement implémentée à l'aide d'un arbre binaire de recherche, dont les feuilles sont les arcs de la rive et les noeuds internes sont les intersections entre les paraboles. Cependant, il reste nécessaire de garder une structure de liste doublement chaînée sur les feuilles de l'arbre, car la gestion des évènements nécessite de connaître les voisins gauche et droit d'un arc donné, c'est pourquoi j'ai fait le choix de l'implémenter à l'aide d'une *liste à enjambement* (*skiplist*).

4.2.1 Structure de liste à enjambement

Ici, un chaînon de la liste représentera donc un arc de la ligne de rive, et un lien entre deux chaînons consécutifs représentera l'intersection entre leurs paraboles respectives, et les arcs seront alors ordonnés par leur position dans la ligne de rive.

L'idée derrière le principe de *skiplist* est d'accélérer la recherche d'un élément en ayant la possibilité de "sauter" par-dessus certains chaînons.

La liste est organisée en *niveaux* : tous les chaînons sont présents au niveau 0, et un chaînon a une chance sur 2 d'être au niveau 1, 1 sur 4 d'être au niveau 2, etc. . De plus, si un chaînon apparait à un niveau, il est également présent sur tous les niveaux inférieurs. Ainsi, on effectuera une recherche à partir du plus haut niveau, et en navigant sur un niveau jusqu'à ce que l'on risque de dépasser l'objectif, auquel cas on passe au niveau inférieur et on continue la recherche.

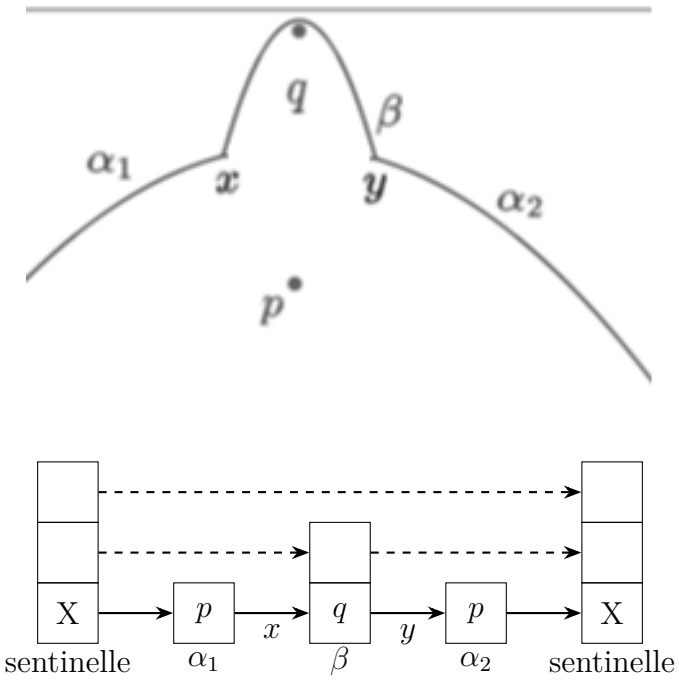


FIGURE 6 – Liste à enjambement représentant L

4.2.2 Complexité moyenne

Considérons une liste de taille n.

On note X_i la variable aléatoire donnant le niveau maximal auquel est présent le i-ème chaînon. En particulier, les X_i sont indépendantes, et suivent toutes une loi géométrique de paramètre $1/2$.

Lemme 4.1. L'espérance de la hauteur maximale d'un chaînon de L est en $O(\ln n)$

Preuve. Notons M la variable aléatoire donnant la hauteur maximale d'un chaînon.

$$\mathbb{E}(M) = \sum_{k=1}^{\infty} \mathbb{P}(M \geq k)$$

$$
\begin{aligned}
&\leq \sum_{k=0}^{\infty} \sum_{m=k\lfloor \ln n \rfloor}^{(k+1)\lfloor \ln n \rfloor - 1} \mathbb{P}(M \geq k) \\
&\leq \sum_{k=0}^{\infty} \sum_{m=k\lfloor \ln n \rfloor}^{(k+1)\lfloor \ln n \rfloor - 1} \frac{1}{2^k} \\
&\leq \sum_{k=0}^{\infty} \sum_{m=k\lfloor \ln n \rfloor}^{(k+1)\lfloor \ln n \rfloor - 1} \frac{1}{2^{k \ln n}} \\
&\leq \left(\sum_{k=0}^{\infty} \frac{1}{2^{k \ln n}} \right) \ln n \\
&\leq \frac{1}{1 - \frac{1}{n^{\ln 2}}} \ln n = O(\ln n)
\end{aligned}
$$

Lemme 4.2. La complexité moyenne d'une recherche dans L est en $O(\ln n)$

Preuve. On considère le chemin inverse effectué pour la recherche d'un élément dans la liste.

Soit C la variable aléatoire correspondant au nombre de chainons visités pendant ce trajet, et C_k celle correspondant à ceux visités au niveau k.

Alors :

$$
\begin{aligned}
\mathbb{E}(C_k) &= \sum_{m=1}^{\infty} k \cdot \mathbb{P}(C_k = m) \\
&= \sum_{m=1}^{\infty} m \cdot \mathbb{P}(C_k = m | C_k \geq 1) \cdot \mathbb{P}(C_k \geq 1) \\
&\leq \sum_{m=1}^{\infty} m \cdot \mathbb{P}(C_k = m | C_k \geq 1) \cdot \mathbb{P}(M \geq k) \\
&\leq \sum_{m=1}^{\infty} \frac{m}{2^m} \cdot \mathbb{P}(M \geq k) \\
&\leq A \cdot \mathbb{P}(M \geq k)
\end{aligned}
$$

Ainsi,

$$
\begin{aligned}
\mathbb{E}(C) &= \sum_{k=0}^{\infty} \mathbb{E}(C_k) \\
&\leq \sum_{k=0}^{\infty} A \cdot \mathbb{P}(M \geq k) \\
&\leq A \cdot \mathbb{E}(M)
\end{aligned}
$$

D'où le résultat d'après le lemme 4.1 .

4.3 Analyse de complexité

Lemme 4.3. Le nombre d'évènements entrants et sortants de Q est en $O(n)$

Preuve.

Un évènement présent à un moment dans Q est soit un *circle event* soit un *site event*.

Il y a précisément n *site events* par définition.
Un circle event est :

— Soit un sommet du diagramme : Il y en a au plus $2n - 4$ d'après le lemme 2.1
— Soit annulé à un moment : Par la découverte d'un point (celle-ci annule au plus un cercle) ou par la confirmation d'un autre *circle event* (qui annule au plus 2 cercles). Il y en a donc au plus $5n - 8$.

D'où le résultat.

Lemme 4.4. La taille de L est en $O(n)$

Preuve. Puisque les chaînons représentent chacun une intersection entre 2 morceaux de parabole de la *beachline*, et que ceux-ci tracent les arêtes du diagramme, qui sont en nombre $O(n)$ d'après le lemme 2.1 on en déduit que la taille de L est bien en $O(n)$

Théorème 4.1. La complexité temporelle de l'algorithme de Fortune est $O(n \ln n)$

Preuve Considérons une itération de la boucle **tant que** de l'algorithme de Fortune.
On dépile un évènement de Q : l'opération est en $O(\ln |Q|) = O(\ln n)$

— Si c'est un *site event*
  — Recherche de a_0 dans L : $O(\ln |L|) = O(\ln n)$ en moyenne
  — Annulation de son circle event : $O(1)$
  — Insertion des deux nouveaux noeuds dans la liste : $O(\ln n)$
  — Calcul des nouveaux évènements : $O(1)$
  — Ajout des arêtes nécessaires : $O(1)$
  — Deux insertions dans Q : $O(\ln n)$
— Si c'est un *circle event*
  — Calcul des nouveaux évènements : $O(1)$
  — Mise à jour et ajout des arêtes : $O(1)$
  — Supression dans L : $O(\ln n)$ en moyenne

La complexité d'une itération est donc en moyenne en $O(\ln n)$. Or, d'après le lemme 4.1 on effectuera $O(n)$ itérations de la boucle, ce qui nous permet d'obtenir une complexité temporelle de l'algorithme en $O(n \ln n)$.

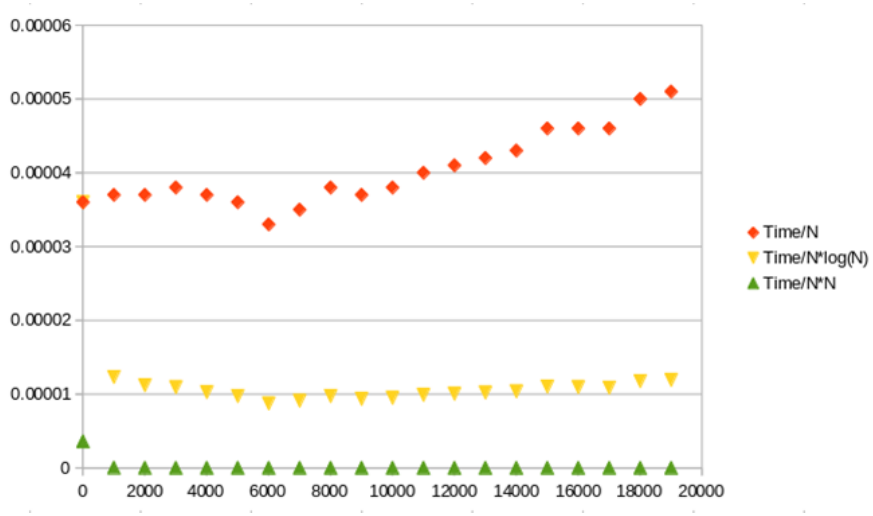
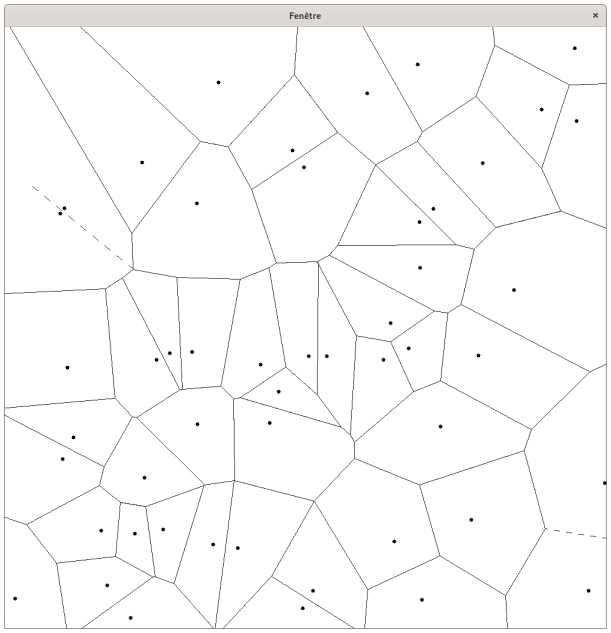
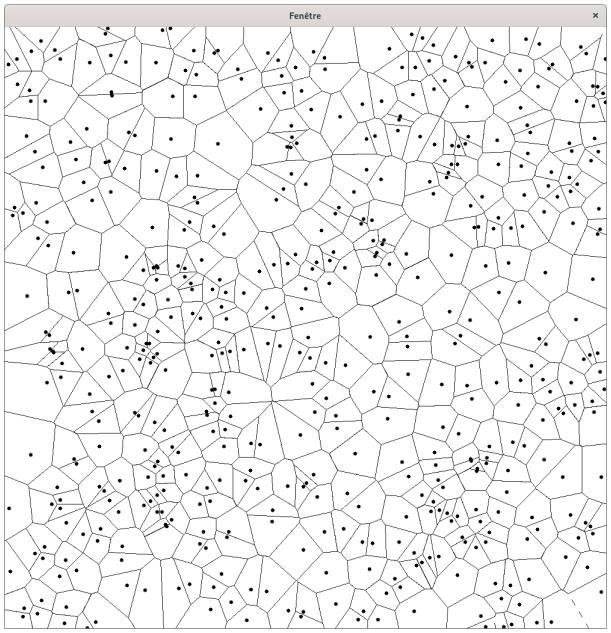


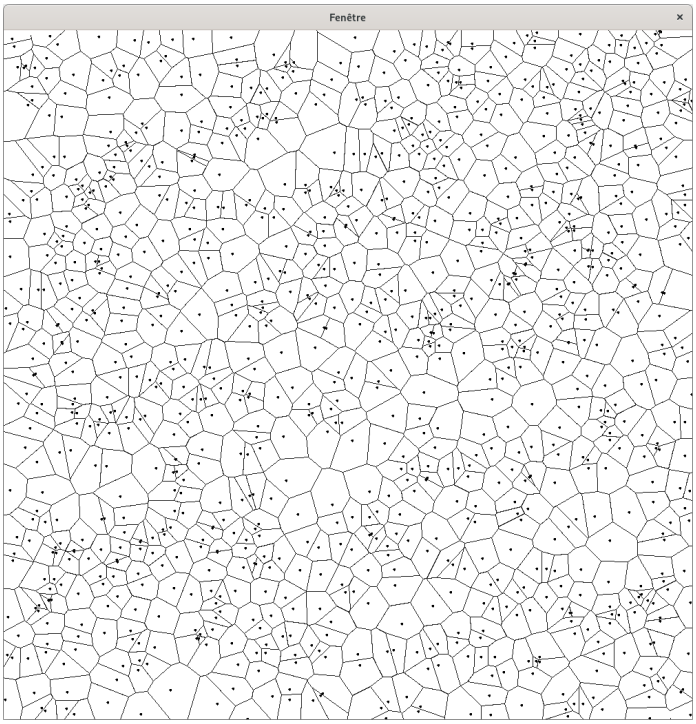
FIGURE 7 – Temps d'éxécution obtenus



$n = 50$



$n = 500$



$n = 1000$

FIGURE 8 – Diagrammes construits pour $n \in \{50, 500, 1000\}$.

5 Annexes

```
pointf intersection(pointf p0, pointf p1, double lgn) {
    pointf res = p0;
    pointf p = p0;
    double r = 0;
    if (feg(p0.y, p1.y)) {
        res.x = (p0.x + p1.x) / 2;
    } else if (feg(p1.y, lgn)) {
        res.x = p1.x;
    } else if (feg(p0.y, lgn)) {
        res.x = p0.x;
        p = p1;
    } else {
        double z0 = 2 * (p0.y - lgn);
        double z1 = 2 * (p1.y - lgn);

        double a = 1 / z0 - 1 / z1;
        double b = -2 * (p0.x / z0 - p1.x / z1);
        double c = (p0.x * p0.x + p0.y * p0.y - lgn * lgn) / z0 - (
           p1.x * p1.x + p1.y * p1.y - lgn * lgn) / z1;

        r = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
        res.x = r;
    }
    res.y = (p.y * p.y + (p.x - res.x) * (p.x - res.x) - lgn * lgn)
       / (2 * p.y - 2 * lgn);
    if (res.y > MAX) {
        res.y = MAX;
    }
    if (res.y < -MAX) {
        res.y = -MAX;
    }
    return res;
}
```

FIGURE 9 – Calcul du breackpoint entre 2 paraboles
(Adapté de `https://www.cs.hmc.edu/~mbrubeck/voronoi.html`)

```
    evenement *converge(pointf p0, pointf p1, pointf p2, double lgn)
        {
    if (p0.x == p1.x || p1.x == p2.x || p2.x == p0.x) {
        return NULL;
    }
    //Methode du determinant pour le centre d'un cercle circonscrit
    double det1 = det3x3(p0.x * p0.x + p0.y * p0.y, p0.y, 1.0,
                         p1.x * p1.x + p1.y * p1.y, p1.y, 1.0,
                         p2.x * p2.x + p2.y * p2.y, p2.y, 1.0);

    double det2 = det3x3(p0.x, p0.x * p0.x + p0.y * p0.y, 1.0,
                         p1.x, p1.x * p1.x + p1.y * p1.y, 1.0,
                         p2.x, p2.x * p2.x + p2.y * p2.y, 1.0);

    double det3 = det3x3(p0.x, p0.y, 1.0,
                         p1.x, p1.y, 1.0,
                         p2.x, p2.y, 1.0);

    if (fabs(det3) < EPSILON) {
        return NULL;
    }
    double x = det1 / (2 * det3);
    double y = det2 / (2 * det3);
    double prio = (y + sqrt((x - p0.x) * (x - p0.x) + (y - p0.y) * (
       y - p0.y)));
    pointf p;
    pointf dq1 = intersection(p0, p1, prio);
    p.x = x;
    p.y = y;
    //Le circle event est en dessous de la ligne de rive (avec marge
        d'erreur) et les breakpoints convergent bien vers le centre
       du cercle lorsque la ligne avance
    if (prio <= lgn + EPSILON || (distance(dq1, p) > EPSILON)) {
        return NULL;
    }
    evenement *ans = malloc(sizeof(evenement));
    ans->est_cercle = true;
    ans->est_site = false;
    ans->p.x = x;
    ans->prio = prio;
    ans->p.y = y;

    ans->ind = -1;

    return ans;
}
```

FIGURE 10 – Calcul du circle event potentiel associé à 3 points

```
    //ind : indice du nouveau site
    //courant : arc sous le nouveau site
    //new1 : arc cree par le nouveau site
    demi_arete *b1 = arete(ind);
    demi_arete *b2 = arete(courant->ind);
    b1->debut = intersection(courant->p, new1->p, lgn + 10.0);
    b2->debut = intersection(new1->p, new2->p, lgn + 10.0);
    new2->arete = b2;
    new1->arete = b1;
    b1->jumelle = b2;
    b2->jumelle = b1;
    cellules[ind] = b1;
    if (!cellules[new2->ind]) {
        cellules[new2->ind] = b2;
    }
    b1->ind = ind;
    b2->ind = new2->ind;
    *DCEL = cons((void *) b1, *DCEL);
    *DCEL = cons((void *) b2, *DCEL);
```

FIGURE 11 – Ajout des arêtes dans GèreSiteEvent

```
1       bool cont = true;
2       while (cont) {
3           if (niveau < 0) {
4               if (courant->sentinelle && courant->suiv) {
5
6                   courant = courant->suiv;
7                   for (int i = 0; i < courant->hauteur; i++) {
8                       if (i < new1->hauteur) {
9                           a_modifier1[i] = courant;
10                      } else if (i < new2->hauteur) {
11                          a_modifier2[i] = courant;
12                      }
13                  }
14              }
15              while (apres(p, courant, lgn)) {
16                  courant = courant->suiv;
17              }
18
19              cont = false;
20          } else {
21              if (!apres(p, courant->skip[niveau]->prec, lgn)) {
22                  if (niveau < new1->hauteur) {
23                      a_modifier1[niveau] = courant;
24                  } else if (niveau < new2->hauteur) {
25                      a_modifier2[niveau] = courant;
26                  }
27                  niveau--;
28              } else {
29
30                  courant = courant->skip[niveau];
31              }
32          }
33      }
```

FIGURE 12 – Recherche de l'arc sous p dans GèreSiteEvent

```
    demi_arete *d1 = c->arete;
    demi_arete *d2 = c->suiv->arete;

    d1->debut = p;
    d2->debut = p;
    d1->complete = true;
    d2->complete = true;
    demi_arete *new1 = arete(c->prec->ind);
    demi_arete *new2 = arete(c->suiv->ind);
    new1->jumelle = new2;
    new2->jumelle = new1;
    new1->debut = p;
    new2->debut = intersection(c->prec->p, c->suiv->p, lgn + 10.0);
    relie(d2->jumelle, d1);
    relie(d1->jumelle, new1);
    relie(new2, d2);
    c->suiv->arete = new2;
    *DCEL = cons((void *) new1, *DCEL);
    *DCEL = cons((void *) new2, *DCEL);
```

FIGURE 13 – Ajout des arêtes dans GèreCircleEvent

Références

[1] Franck Hétroy, *Un petit peu de géométrie algorithmique.*

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars, *Computational Geometry : Algorithms and Applications*, Third Edition, Springer, ISBN 978-3-540-77973-5, DOI : 10.1007/978-3-540-77974-2, Chapitre 7.

[3] Steven Fortune, "A sweepline algorithm for Voronoi diagrams", *Algorithmica* 2, 153–174 (1987).

[4] pvigier's blog, *Fortune's Algorithm : The Details*, https://pvigier.github.io/2018/11/18/fortune-algorithm-details.html.

[5] Tufts University, *Fortune's Algorithm Visualization.* https://www.cs.tufts.edu/comp/163/demos/fortune/

[6] Johannes Jensen, *Computing Voronoi Diagrams Using Fortune's Algorithm.*

[7] François Schwarzentruber, *Skip lists - Listes à saut*

Certaines des images présentes dans ce rapport ont été adaptées de la visualisation proposée par [5]