

**Filière MP - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2022**  
**Page de garde du rapport de TIPE**

NOM : FRADIN	Prénoms : Adrien, Laurent
Classe : MP1	
Lycée : Lycée Michel - Montaigne	Numéro de candidat : 14 560
Ville : Bordeaux	

Concours auxquels vous êtes admissible, dans la banque MP Inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP	<input type="checkbox"/>	MP - Option MPI	<input checked="" type="checkbox"/>
	Informatique	<input type="checkbox"/>		<input type="checkbox"/>
ENS Lyon	MP - Option MP	<input type="checkbox"/>	MP - Option MPI	<input checked="" type="checkbox"/>
	Informatique - Option M	<input type="checkbox"/>	Informatique - Option P	<input type="checkbox"/>
ENS Rennes	MP - Option MP	<input type="checkbox"/>	MP - Option MPI	<input type="checkbox"/>
	Informatique	<input checked="" type="checkbox"/>		<input type="checkbox"/>
ENS Paris	MP - Option MP	<input type="checkbox"/>	MP - Option MPI	<input checked="" type="checkbox"/>
	Informatique	<input type="checkbox"/>		<input type="checkbox"/>

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique <input checked="" type="checkbox"/>	Mathématiques <input type="checkbox"/>	Physique <input type="checkbox"/>
--	--	-----------------------------------

Titre du TIPE :

Confinement localisé du centre-ville de Bordeaux

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	6	Illustration	8	Bibliographie	1/2
-------	---	--------------	---	---------------	-----

Attention, les illustrations doivent figurer dans le corps du texte et non en fin du document !

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Ce travail a pour but de mettre en place les bases d'un modèle de confinement local, en découplant automatiquement une région donnée en zones "autosuffisantes". En se concentrant sur le centre-ville de Bordeaux, nous y construirons un diagramme de Voronoï que nous adapterons aux routes du centre-ville, afin de tenir compte des potentiels obstacles géographiques (fleuves, habitations, ...).

A Bordeaux

Le 10/06/2022

Signature du (de la) candidat(e)

Signature du professeur responsable de  
la classe préparatoire dans la discipline

Cachet de l'établissement

**LYCÉE MICHEL MONTAIGNE**  
 226 rue Sainte Catherine  
 33075 BORDEAUX Cedex

La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).

## Confinement localisé du centre-ville de Bordeaux

par

FRADIN Adrien (n°14560)

(Ce TIPE a fait l'objet d'un travail de groupe dont voici ma partie)

---

## Table des matières

<b>1</b>	<b>Regroupement des commerces de proximité</b>	<b>3</b>
1.1	Extraction des magasins . . . . .	3
1.2	Construction des groupes de magasins avec des arbres 2-D . . . . .	3
1.2.1	Construction d'un arbre 2-D . . . . .	3
1.2.2	Recherche dans un arbre 2-D . . . . .	4
1.2.3	Résultats du regroupement . . . . .	7
<b>2</b>	<b>Construction de diagrammes de Voronoï (via l'algorithme de Fortune)</b>	<b>7</b>
2.1	Définition et premières propriétés . . . . .	7
2.1.1	Structure des diagrammes de Voronoï . . . . .	7
2.1.2	Résultats quantitatifs . . . . .	8
2.2	Implémentation de l'algorithme de Fortune . . . . .	9
2.2.1	Les deux types d'événements . . . . .	9
2.2.2	Modélisation de la ligne de front par un arbre binaire équilibré . . . . .	10
2.2.3	Modélisation du diagramme de Voronoï en construction . . . . .	12
<b>3</b>	<b>Ajustement du tracé des zones aux routes</b>	<b>13</b>
3.1	Construction du graphe des routes . . . . .	13
3.2	Ajustement des arêtes du diagramme de Voronoï aux routes . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>14</b>
<b>5</b>	<b>Annexes</b>	<b>15</b>

## Introduction : motivations & objectifs

Le confinement national en France (suite à la pandémie de Covid-19) n'a guère fait l'unanimité d'autant plus que les restrictions de circulation (de 1 km autour du domicile) ont mis un coup de frein aux commerces et à l'économie : n'aurait-il pas été plus judicieux de mettre en place un confinement *local*,<sup>1</sup> fondé sur un découpage en zones autosuffisantes ? L'idée derrière ce découpage est de ne pénaliser que les *mauvais élèves* (i.e. les zones où l'épidémie se propage) en les confinant, tout en laissant les autres zones ouvertes (commerces ouverts, déplacements entre zones autorisés, levée partielle des restrictions, etc.).<sup>2</sup>

Tout au long de ce TIPE, nous nous focaliserons sur le cas du centre-ville de Bordeaux dont le découpage se fera en trois temps :

- 1) pour que ces zones soient autonomes, elles devront contenir au moins un supermarché et une pharmacie (les commerces *essentiels*) : nous regrouperons donc ces magasins en « cluster »,
- 2) une fois leur contenu fixé, nous délimiterons les frontières de ces zones. L'idée étant que les individus d'une zone soient à plus proche distance d'un magasin de leur zone plutôt que des magasins des zones voisines : nous construirons donc un diagramme de Voronoï.
- 3) enfin, parce que le découpage obtenu en 2) ne tient pas compte de la géographie des lieux (i.e. certaines arêtes peuvent traverser des habitations, des fleuves, des voies ferrées, etc.) il conviendra d'ajuster le tracé de ces zones aux routes du centre-ville.

Derrière l'étude du cas particulier qu'est le centre-ville de Bordeaux, le lecteur constatera que les outils mis en œuvre permettront d'effectuer de tels découpages en zones sur d'autres régions de France, voir à l'échelle nationale ; et ces découpages se sont automatiquement.

Objet principal de ce TIPE, les diagrammes de Voronoï (nommée ainsi en l'honneur du mathématicien russe Gueorgui Voronoï) sont aujourd'hui des structures mathématiques largement répandues<sup>3</sup> et leur utilisation pratique ne date pas d'hier : en 1853 le britannique John Snow, grâce aux diagrammes de Voronoï, a identifié l'origine de l'épidémie de choléra à Londres.

Nous nous concentrerons dans le présent travail à leur construction informatique<sup>4</sup> ainsi qu'à un exemple d'application concrète de ces diagrammes, en lien avec la thématique santé/prévention. Nous nous appuierons principalement sur l'ouvrage de référence *Computational Geometry* [2] (et particulièrement les chapitres 5 et 7) ainsi que sur les chapitres 10 et 17 de [4]. Nous mettrons en œuvre l'algorithme de Fortune, inventé par l'ingénieur S. Fortune en 1987 comme préconisé par [2] (pour son efficacité).

---

1. Local à l'échelle d'une ville/d'un village.

2. La simulation d'une épidémie avec un tel modèle (partie qui suivrait naturellement la conception de ce modèle) a été traité par le camarade avec qui ce TIPE a été élaboré et n'est donc pas évoquée ici.

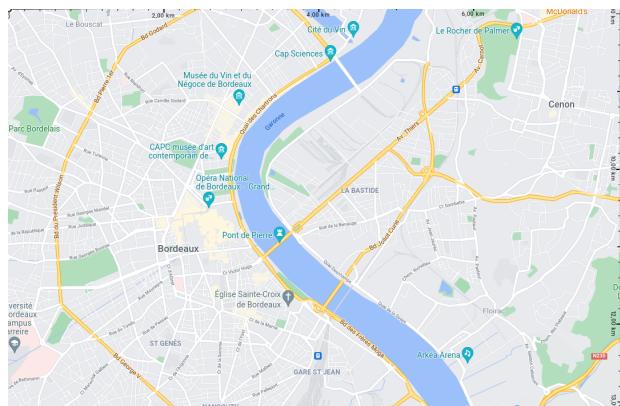
3. Ils se retrouvent, par exemple, dans la nature avec le pelage des girafes.

4. Le langage de programmation utilisé dans ce TIPE est Python.

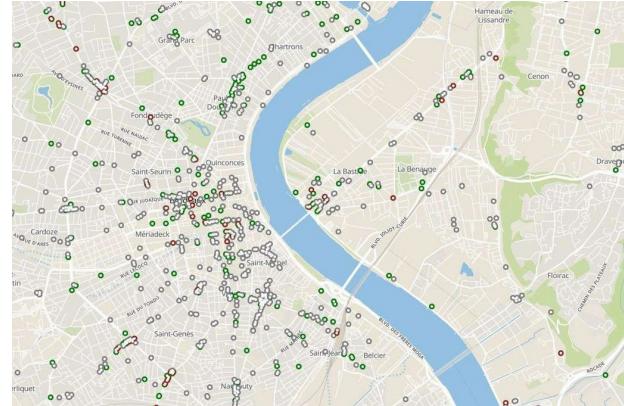
# 1 Regroupement des commerces de proximité

## 1.1 Extraction des magasins

Tout d'abord, il nous faut des données sur la localisation et le type des magasins présents dans le centre-ville de Bordeaux. Nous utilisons pour cela le site *Ça reste ouvert*<sup>5</sup> :



(a) Carte du centre-ville : région étudiée (Google Maps).



(b) Carte du site Ça reste ouvert : un point par magasin.

Chaque ligne du fichier obtenu (au format *csv*) se présente ainsi :

*node/7746394597,Picard,food,supermarket,[...],-0.6136596,44.854885*

FIGURE 2 – Une ligne de la base de donnée dont : le type de magasin, sa longitude et sa latitude.

Nous ne conservons que les supermarchés et pharmacies contenues dans la région étudiée ; grâce au module PIL de Python nous affichons automatiquement ces magasins :

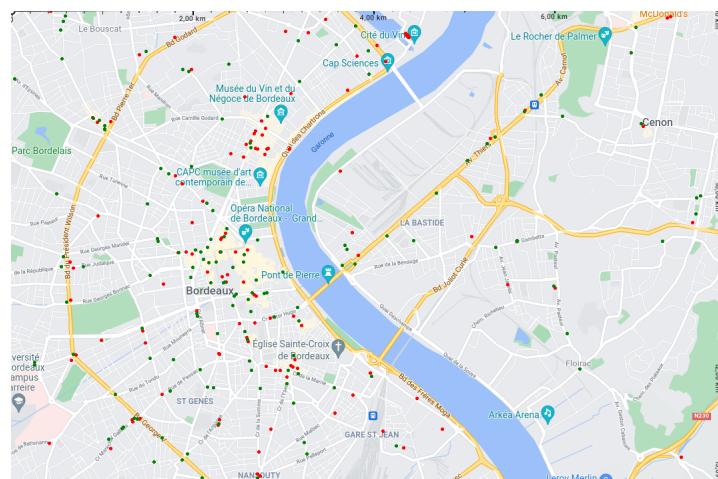


FIGURE 3 – Légende : supermarchés et pharmacies .

## 1.2 Construction des groupes de magasins avec des arbres 2-D

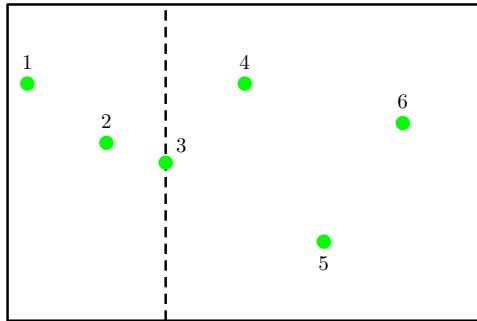
Pour construire ces groupes, nous procédons ainsi : pour chaque supermarché, nous cherchons la pharmacie la plus proche puis, nous les associons. S'il reste des pharmacies encore seules, nous les mettons dans le même groupe que celui du supermarché le plus proche. Cette recherche des plus proches voisins va être effectuée grâce à un arbre 2-D :

**Définition n°1 :** (arbre 2-D) un arbre 2-D est un arbre binaire associé à une partition finie du plan (en rectangles).

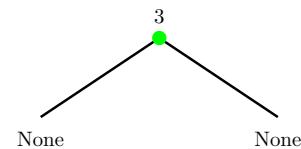
### 1.2.1 Construction d'un arbre 2-D

Étant donné des points du plan, deux-à-deux distincts, nous construisons récursivement un arbre 2-D en séparant l'espace en deux demi-espaces suivant les points fournis, tout en alternant l'axe de coupe (suivant les *x* puis les *y*, les *x* etc.). Voici un exemple illustré :

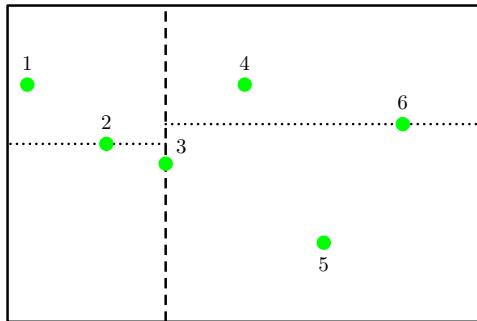
5. La carte provenant du site (cf. image à droite) n'est plus disponible, toutefois les jeux de données le sont encore à l'adresse : <https://www.data.gouv.fr/fr/organizations/ca-reste-ouvert/>



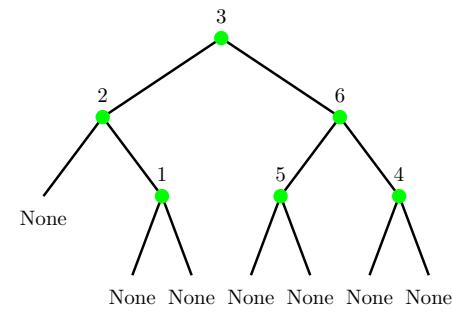
(a) Première coupe suivant 3.



(b) Arbre binaire associé.



(c) Coupes récursives suivant 2 (à gauche) et 6 (à droite).



(d) Arbre 2-D final.

FIGURE 5 – Exemple de construction d'un arbre 2-D : à gauche le découpage du plan, à droite l'arbre binaire associé.

Afin que l'arbre obtenu soit équilibré, nous choisissons le point médian pour la coupe ; ci-dessous, le pseudo-code de la construction :

#### Algorithme 1 : Algorithme de construction d'un arbre 2-D

**Données :** Un tableau  $T$  de points distincts du plan.

**Résultat :** Un arbre 2-D associé à ces points.

**1 Fonction Construire( $m, M, axe$ ) :**

```

1   /* Construire un arbre 2-D associé aux points  $T[m : M]$  suivant l'axe de coupe (1 pour les  $x$ , 0 pour les  $y$ ) */
2   si  $m \geq M$  alors
3       retourner None
4   sinon
5       Trier le sous-tableau  $T[m : M]$  par ordre lexicographique suivant la coordonnée  $axe$ 
6       milieu  $\leftarrow \lfloor \frac{m+M}{2} \rfloor$ 
7       retourner Noeud( $T[milieu]$ , Construire( $m, milieu - 1, 1 - axe$ ), Construire( $milieu + 1, M, 1 - axe$ ))
    
```

On a les résultats suivants (preuves en [annexe](#)) :

**Propriété n°1 :** L'algorithme 1 termine et renvoie un arbre binaire équilibré de hauteur  $1 + \lceil \log_2(n) \rceil$  pour  $n > 0$  points.

**Propriété n°2 :** La complexité spatiale (resp. temporelle) de l'algorithme 1 est un  $O(n)$  (resp.  $O(n \log^2(n))$ ).<sup>6</sup>

#### 1.2.2 Recherche dans un arbre 2-D

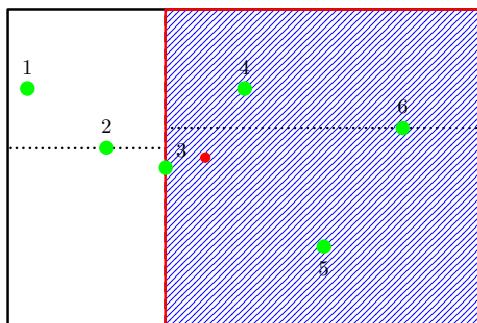
Les arbres 2-D permettent une recherche efficace (par rapport à une approche *naïve*) d'un plus proche voisin en réduisant le nombre de distances à calculer.<sup>7</sup> La recherche d'un plus proche voisin d'un point  $P$  donné alterne entre deux phases :

- 1) la descente : on descend dans l'arbre binaire jusqu'aux feuilles (via des comparaisons lexicographiques) en cherchant la région la *plus proche* du point  $P$ .
- 2) la remontée : on remonte au noeud parent puis on vérifie si un plus proche voisin peut se trouver dans le fils non visité, auquel cas (nous parlerons de *basculement*), on recommence 1 depuis ce fils ; sinon, on remonte encore.

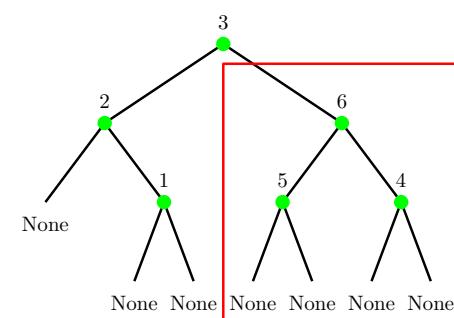
Voisin un exemple illustré, étant donné un point rouge  $P$ , on souhaite trouver parmi les points verts, un plus proche voisin de  $P$  :

6. Nous effectuons la sélection du point médian (ligne 5) via un tri fusion du tableau des points. Nous tenons à informer le lecteur de l'existence d'algorithme de sélection de la médiane en temps linéaire, abaissant la complexité temporelle de la construction à un  $O(n \log(n))$ .

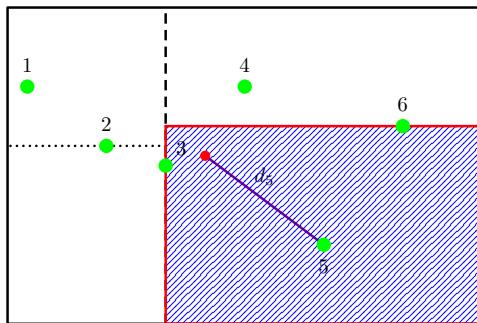
7. Typiquement, on exclut certaines des régions (construites précédemment) de la recherche.



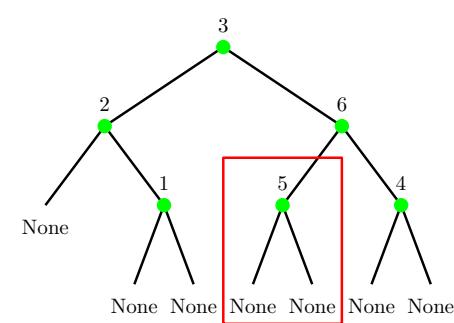
(a) Le point rouge  $P$  est localisé à droite de 3.



(b) On descend alors dans le fils droit de 3.



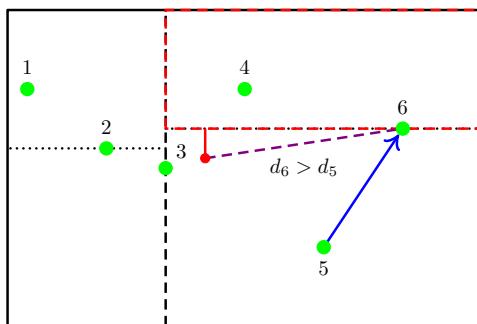
(c) On mémorise le point 5 et la distance  $d_5 < +\infty$ .



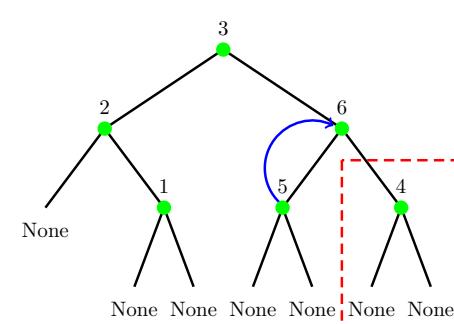
(d) On aboutit finalement dans le fils gauche de 6.

FIGURE 7 – Phase de descente.

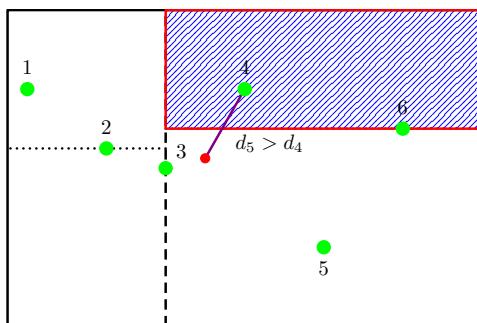
On remonte alors dans l'arbre, au nœud parent qui est 6 :



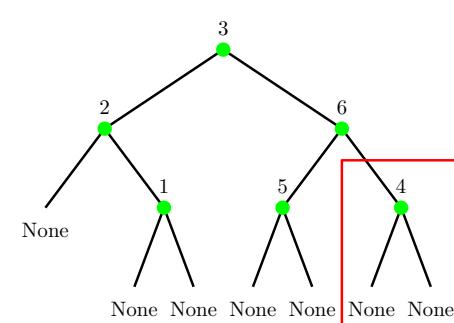
(a) On met à jour, si besoin, le plus proche voisin.



(b) On vérifie s'il y a basculement dans le fils droit de 6.



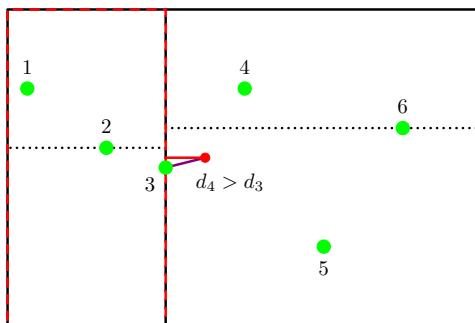
(c) Il faut basculer : on recommence l'étape 1.



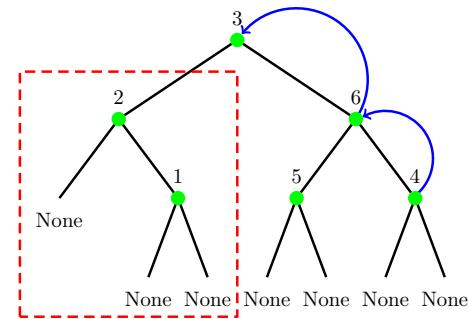
(d) On trouve que 4 est plus proche de  $P$  que ne l'était 5.

FIGURE 9 – Phase de remontée.

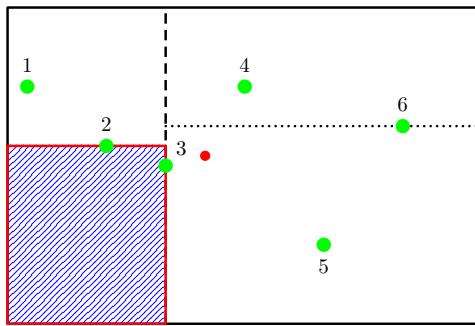
On continue la phase de remontée :



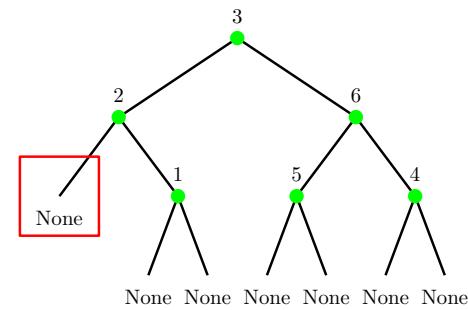
(a) On met à jour le plus proche voisin de  $P$  : c'est 3.



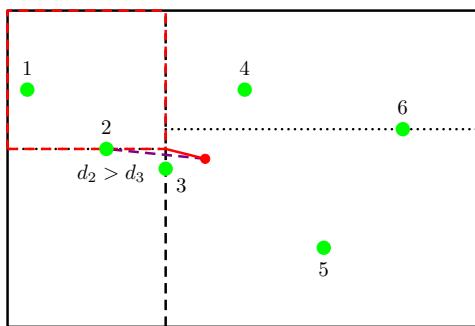
(b) On vérifie s'il y a basculement dans le fils droit de 3.



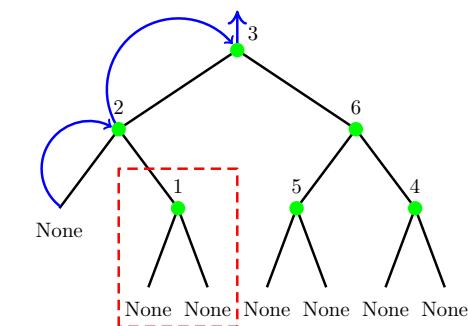
(c) Il faut basculer : on recommence l'étape 1.



(d) Le sous-arbre est vide : on remonte directement au nœud 2.



(e) Pas de basculement dans le fils droit de 2 : on remonte.



(f) On sort de l'arbre : fin de la recherche.

et le programme renvoie 3 comme plus proche voisin de  $P$ .

Le test de basculement consiste à calculer la distance entre le point  $P$  et la région délimitée par le sous-arbre concerné et permet de vérifier s'il peut y avoir des points plus proches de  $P$  parmi ceux du fils non visité ; voici le pseudo-code de la recherche dans un arbre 2-D :

#### Algorithme 2 : Algorithme de recherche dans un arbre 2-D

**Données :** Un arbre 2-D et un point  $P$ .

**Résultat :** Un plus proche voisin sous forme d'un couple de coordonnées  $(x, y)$ .

**1 Fonction Rechercher(arb, axe) :**

```

1  /* Recherche un plus proche voisin du point P dans l'arbre 2-D arb suivant l'axe de coupe. */ 
2  si arb ≠ None alors
3      si EstFeuille(arb) alors
4          MettreAJour(arb) // Mettre à jour, si besoin, le plus proche voisin et sa distance.
5      sinon si DescendreGauche(arb, axe, P) alors
6          Rechercher(FilsGauche(arb), 1 - axe)
7          MettreAJour(arb)
8          si Basculer(arb, FilsDroit(arb)) alors
9              Rechercher(FilsDroit(arb), 1 - axe)
10     sinon // Descente dans le fils droit de l'arbre.
11         Rechercher(FilsDroit(arb), 1 - axe)
12         MettreAJour(arb)
13         si Basculer(arb, FilsGauche(arb)) alors // Vérifier s'il faut basculer dans le fils gauche.
14             Rechercher(FilsGauche(arb), 1 - axe)
    
```

On a (preuve en [annexe](#)) :

**Propriété n°3 :** *L'algorithme 2 termine et renvoie un plus proche voisin de P.*

**Propriété n°4 :** *La complexité spatiale (resp. temporelle) de l'algorithme 2 est un O(1) (resp. O(n log(n)) en moyenne et un O( $n^2$ ) dans le pire cas<sup>8</sup>).*

*Preuve :* admise (le lecteur intéressé trouvera une preuve dans [3]).

### 1.2.3 Résultats du regroupement

Finalement, après regroupement, nous obtenons la carte suivante :

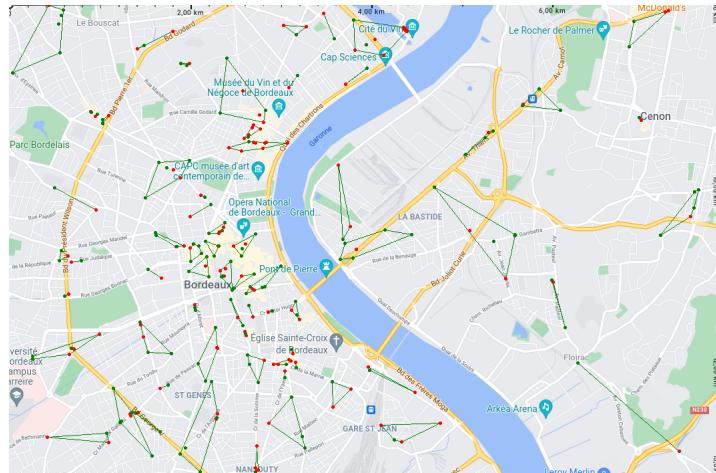


FIGURE 11 – Construction des groupes.

où sont dessinées les enveloppes convexes des groupes de points (nous avons utilisé l'algorithme d'Andrew [1]).

## 2 Construction de diagrammes de Voronoï (via l'algorithme de Fortune)

### 2.1 Définition et premières propriétés

#### 2.1.1 Structure des diagrammes de Voronoï

On considère  $n$  points  $P_1, \dots, P_n$  du plan deux-à-deux distincts (appelés sites) en position générale<sup>9</sup> :

**Définition n°2 :** (cellule de Voronoï) Soit  $i \in \llbracket 1; n \rrbracket$ , la cellule de Voronoï associée au point  $P_i$  est :

$$C_i = \{Q \in \mathbb{R}^2 / \forall j \in \llbracket 1; n \rrbracket, d(Q, P_i) \leq d(Q, P_j)\}$$

Les points  $P_1, \dots, P_n$  étant fixés, le diagramme de Voronoï (ensemble des cellules  $C_1, \dots, C_n$ ) est unique. On a :

**Propriété n°5 :** Pour  $i \in \llbracket 1; n \rrbracket$ , la cellule  $C_i$  est un polygone convexe dont :

1) chaque arête est une portion de la médiatrice entre  $P_i$  et un site « voisin »,

2) les sommets sont à équidistance de 3 sites (dont  $P_i$ ).

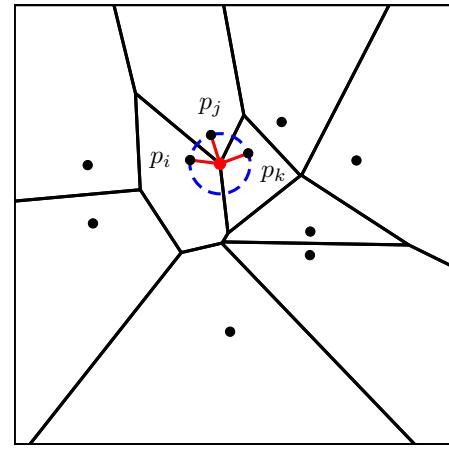
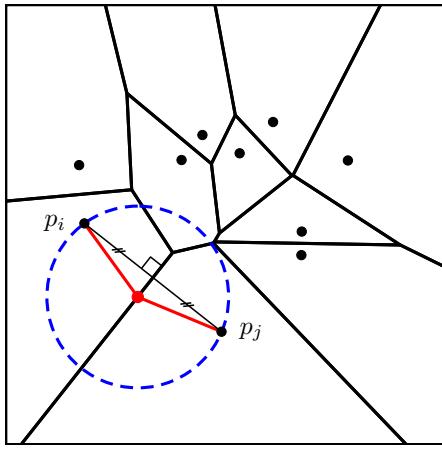
*Preuve :* Pour  $(P, Q) \in \mathbb{R}^2$ , notons  $H(P, Q)$  le demi-espace fermé (convexe) contenant  $P$  et délimité par la médiatrice du segment  $[P, Q]$  alors,  $C_i = \bigcap_{\substack{j=1 \\ j \neq i}}^n H(P_i, P_j)$  est donc un polygone convexe fermé. La frontière de  $C_i$  est  $\text{Fr}(C_i) =$

$\{Q \in C_i / \exists j \in \llbracket 1; n \rrbracket \setminus \{i\} / d(Q, P_i) = d(Q, P_j)\}$  : c'est une réunion de segments et de demi-droites, chacune faisant partie de la médiatrice entre  $P_i$  et un site voisin. Enfin, les sommets de  $C_i$  étant à l'intersection de deux de ses arêtes i.e. de deux médiatrices (d'après ce qui précède) alors, chacun d'eux est à équidistance entre  $P_i$  et deux autres sites distincts.

Illustrons ceci en images :

8. Le pire cas correspond à la situation où tous les noeuds de l'arbre sont visités.

9. Nous ne travaillerons pas dans le cas où les sites sont alignés : c'est la situation dans laquelle les cellules sont des bandes délimitées par deux droites. On parlera de diagramme de Voronoï dégénéré dans ce cas (le lecteur constatera qu'un diagramme de Voronoï est dégénéré si il contient une droite).



(a) Les points d'une arête sont sur la médiatrice de deux sites. (b) Chaque sommet d'une cellule et à équidistance de trois sites.

FIGURE 13 – Structures des cellules de Voronoï.

Dans la suite, nous désignerons les arêtes (resp. sommets) du diagramme de Voronoï comme les arêtes (resp. sommets) des cellules de Voronoï ; ces composantes sont caractérisées par (cf. preuve en [annexe](#)) :

**Propriété n°6 :** Soit  $Q \in \mathbb{R}^2$  alors :

- 1)  $Q$  est sur une arête du diagramme de Voronoïssi il existe deux sites distincts  $P_i$  et  $P_j$  tels que  $r = d(Q, P_i) = d(Q, P_j)$  et pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $P_m \notin B(Q, r)$ .<sup>10</sup>
- 2)  $Q$  est un sommet du diagramme de Voronoïssi il existe trois sites distincts  $P_i$ ,  $P_j$  et  $P_k$  tels que  $r = d(Q, P_i) = d(Q, P_j) = d(Q, P_k)$  et pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $P_m \notin B(Q, r)$ ,

### 2.1.2 Résultats quantitatifs

Le résultat suivant sera utile pour établir les complexités spatiale et temporelle de l'algorithme de construction de diagrammes de Voronoï :

**Propriété n°7 :** Avec  $a$  (resp.  $s$ ) le nombre de d'arêtes (resp. sommets) du diagramme de Voronoï :

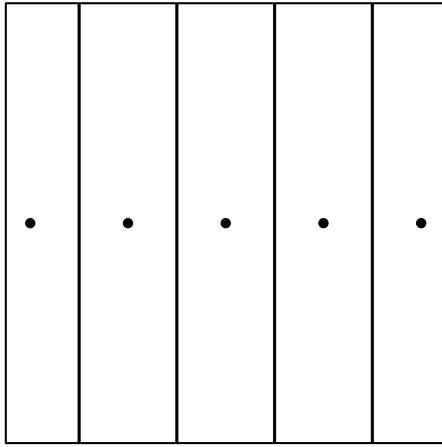
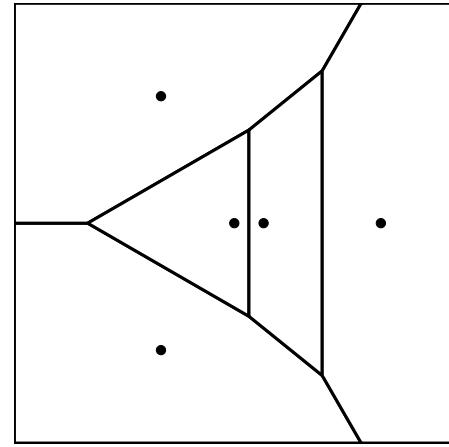
$$n - 1 \leq a \leq 3n - 6 \text{ et } 0 \leq s \leq 2n - 5$$

*Preuve :* (cf. [2]) On crée un sommet à l'infini  $v_\infty$  auquel sont reliées les extrémités de chaque demi-droite. Le graphe obtenu est planaire et connexe<sup>11</sup>, la relation d'Euler donne :

$$(1) : (s + 1) - a + n = 2$$

car chaque face est associée à un unique site. Les sommets de ce graphe étant de degré supérieur ou égal à 3 on en tire  $2a \geq 3(s + 1)$  puis, en injectant dans (1) on obtient  $0 \leq s \leq 2n - 5$  et  $0 \leq a \leq 3n - 6$ . La minoration  $n - 1 \leq a$  découle de (1) avec  $s \geq 0$ .

On notera que ces bornes sont atteintes :

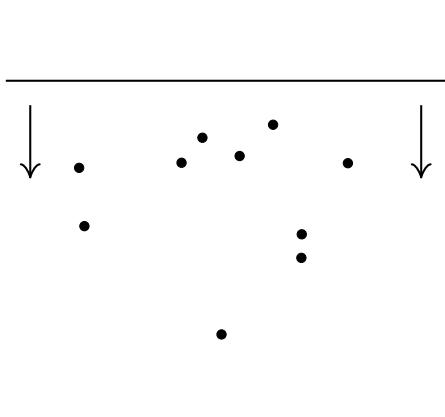
(a) Diagramme de Voronoï dégénéré (ici,  $n = 5$ ,  $s = 0$ ,  $a = 4$ ).(b) Cas où  $a$  et  $s$  sont maximales (ici  $n = 5$ ,  $s = 5$ ,  $a = 9$ ).

10. Boule ouverte de centre  $Q$  et de rayon  $r$ .

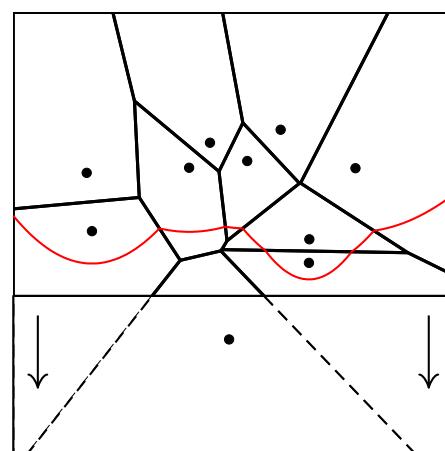
11. En effet, la connexité s'obtient en remarquant, d'une part que nous pouvons toujours joindre deux sommets d'une même cellule et, d'autre part, que les cellules du diagramme de Voronoï sont accolées les unes aux autres. Pour la planéité du graphe, on notera que si deux arêtes du diagramme de Voronoï se croisent, leur point d'intersection est nécessairement un sommet du diagramme de Voronoï.

## 2.2 Implémentation de l'algorithme de Fortune

Nous présentons l'algorithme de Fortune, développé en 1986 par S. Fortune ; il s'agit d'un balayage (ici horizontal) du plan où sont déclenchés des événements suivant leur ordonnée<sup>12</sup>. Voici la droite de balayage (notée  $\ell$  par la suite) :



(a) La droite de balayage (elle descend vers les  $y$  croissants).



(b) On maintient la ligne de front lors du balayage.

L'idée de Fortune est de regarder les points du plan à équidistance entre la droite de balayage et un site au-dessus de cette droite. Ceci forme un ensemble de paraboles (une pour chaque site) dont l'enveloppe inférieure (dessinée en rouge ci-dessus) constitue la ligne de front (notée  $\mathfrak{F}$  désormais).

**Remarque :** *La ligne de front  $\mathfrak{F}$  est caractérisée par : tout point au-dessus de  $\mathfrak{F}$  est plus proche d'un site au-dessus de  $\ell$  que de la ligne de front.*

Nous tâcherons désormais de saisir l'invariant suivant :

**Invariant :** *Au-dessus de la ligne de front se trouve le diagramme de Voronoï (partiel).*

### 2.2.1 Les deux types d'événements

La ligne de front, qui balaye aussi le plan, est composée d'arcs de paraboles et de *breakpoints* (point d'intersection entre deux arcs de paraboles consécutifs). À chaque arc correspond un site (le foyer de la parabole dont fait partie l'arc) ; tout arc sera représenté par un objet Python `Arc` auquel nous lui associons (s'ils existent) ses deux arcs voisins (gauche et droit) ainsi que les deux breakpoints qui « bordent » cette arc.

Chaque *breakpoint* étant à équidistance de deux sites  $P_i \neq P_j$ , il longe une portion de la médiatrice du segment  $[P_i P_j]$  ; nous représentons tout breakpoint par l'objet `BreakPoint` auquel est associé les deux sites  $P_i$  et  $P_j$ , sa direction (un vecteur de  $\mathbb{R}^2$ ) et son « sens » (gauche '`g`' ou droit '`d`'<sup>13</sup>).

**Propriété n°8 :** *Chaque breakpoint trace une partie d'une arête du diagramme de Voronoï.*

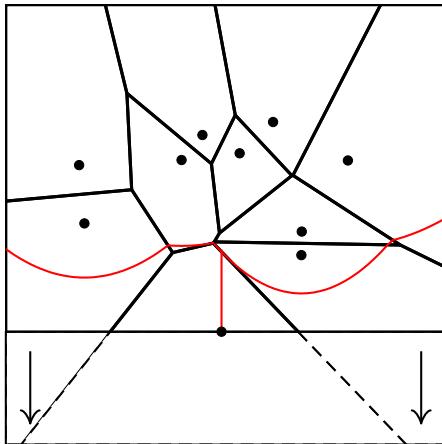
**Preuve :** Soit  $B$  un breakpoint, il est à équidistance de deux sites distincts  $P_i$  et  $P_j$  ainsi que de  $\ell$  (la droite de balayage). Comme  $B$  est un point de l'enveloppe inférieure, pour tout  $k \in \llbracket 1; n \rrbracket$ ,  $r = d(B, P_k) \leq d(B, P_i)$  donc,  $B(B, r)$  ne contient aucun site : d'après la **propriété 6**,  $B$  est sur une arête du diagramme de Voronoï.

Au cours du balayage, on rencontre deux type d'événements :

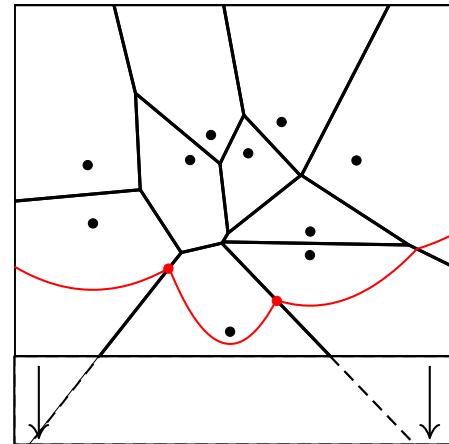
- 1) **les événements ponctuels :** lorsque la droite  $\ell$  passe par un site, un nouvel arc de parabole apparaît dans la ligne de front, tel qu'illustré :

12. Le module PIL de Python impose que le coin supérieur gauche de l'image soit l'origine du repère, l'axe des abscisses est dirigée vers la droite, celui des ordonnées vers le bas.

13. Il est possible que le breakpoint descende verticalement, auquel cas les sens gauche et droit n'ont pas d'importance.



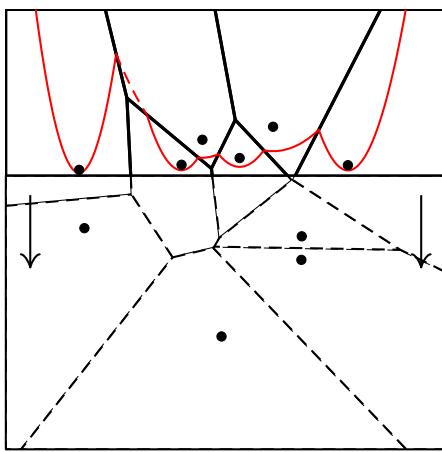
(a) Rencontre avec un site : apparition d'une parabole dégénérée.



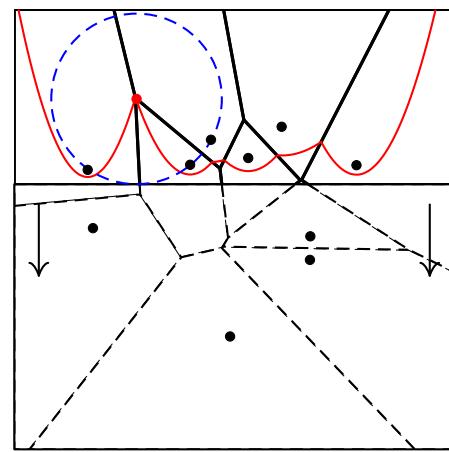
(b) La parabole s'ouvre de plus en plus.

**Remarque :** À un événement ponctuel, deux breakpoints divergeant<sup>14</sup> apparaissent : l'un va à gauche et l'autre à droite. Ces breakpoints, dit jumeaux, tracent ensemble une arête complète du diagramme de Voronoï.

- 2) les événements circulaires : lorsqu'un arc est réduit à un point i.e. lorsque deux breakpoints coïncident, un événement circulaire est déclenché :



(a) L'arc en pointillé va disparaître.



(b) On a détecté un sommet du diagramme de Voronoï.

**Remarque :** Les événements circulaires n'étant pas connus à l'avance, ils sont détectés en regardant si deux breakpoints consécutifs de  $\mathfrak{F}$  convergent i.e. sont susceptible de se rencontrer.

Voici le pseudo-code pour tester si deux breakpoints  $B_1$  et  $B_2$  convergent :

**Algorithme 3 :** Algorithme pour vérifier la convergence de deux breakpoints.

1 Fonction VerifierConvergence( $B_1, B_2$ ) :

/\* On suppose que Direction( $B_1$ ) et Direction( $B_2$ ) sont non colinéaires. \*/

```

2    $P \leftarrow \text{IntersectionDroites}(B_1, \text{Direction}(B_1), B_2, \text{Direction}(B_2))$ 
3    $bool_1 \leftarrow \text{SontPositivementLies}(\text{Vecteur}(B_1, P), \text{Direction}(B_1))$ 
4    $bool_2 \leftarrow \text{SontPositivementLies}(\text{Vecteur}(B_2, P), \text{Direction}(B_2))$ 
5   retourner  $bool_1$  et  $bool_2$ 
```

\*/

Pour déclencher ces événements correctement, on utilise une file de priorité, implémentée via un tas binaire min<sup>15</sup> (non détaillé ici). Pour chaque élément du tas, sa priorité est l'ordonnée à laquelle se déclenche l'événement.

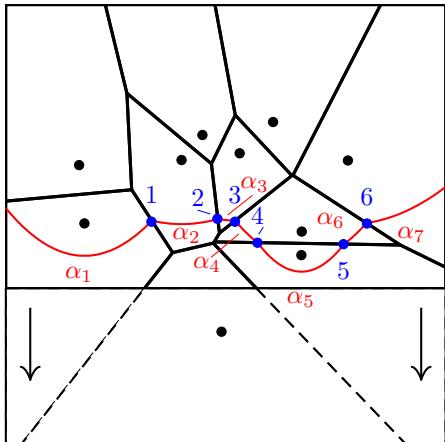
### 2.2.2 Modélisation de la ligne de front par un arbre binaire équilibré

La ligne de front est modélisée par un arbre binaire de recherche équilibré<sup>16</sup> (structure de donnée préconisée dans [2]). Cette structure de donnée, classique, à l'avantage de garantir une recherche, une insertion et une déletion en temps logarithmique en le nombre de noeuds de l'arbre. Voici un exemple de ligne de front et son arbre associé :

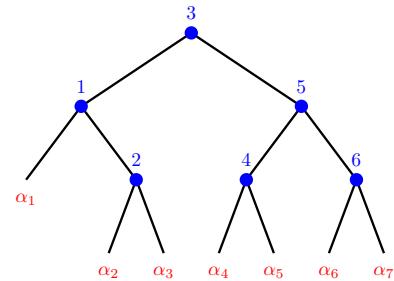
14. Ces breakpoints ne se rencontreront pas.

15. Le lecteur constatera en annexe, la présence d'un booléen *indexation* dans la définition du tas. Ceci permet, sous réserve que les objets présents dans le tas dispose du champs *idx*, de conserver l'indice de l'objet dans le tableau du tas afin de supprimer, si besoin, des objets du tas en temps logarithmique (et d'éviter une recherche linéaire dans le tas).

16. Nous ne détaillerons pas la procédure de ré-équilibrage d'un tel arbre ici, le lecteur pourra consulter en annexe le code concernant le ré-équilibrage.



(a) Ligne de front du diagramme de Voronoï.

(b) Représentation par un arbre binaire équilibré (les feuilles figurent dans le même ordre que dans  $\mathfrak{F}$ ).

Nous imposons :

**Invariant :** Les feuilles de l'arbre représentent les arcs de  $\mathfrak{F}$  et les nœuds internes symbolisent les breakpoints.

La descente dans l'arbre se fait par comparaison lexicographique des coordonnées des breakpointes, qui sont recalculées en temps constant avec les formules :

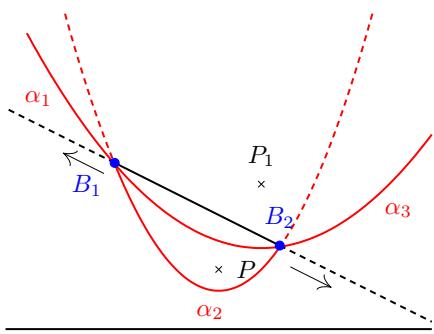
**Propriété n°9 :** Soient  $P_1(x_1, y_1)$  et  $P_2(x_2, y_2)$  deux sites distincts et  $y_D \geq y_1, y_2$  l'ordonnée de la droite  $\ell$  alors, les coordonnées<sup>17</sup> des points d'intersections entre les deux paraboles de foyers  $P_1$  et  $P_2$  et de directrice  $\ell$  sont :

$$\left( \lambda(y_2 - y_1) + \frac{x_1 + x_2}{2}; \lambda(x_1 - x_2) + \frac{y_1 + y_2}{2} \right)$$

où

- si  $y_1 = y_2 \neq y_D$ ,  $\lambda = \frac{1}{2} \left( \frac{y_1 - y_D}{x_2 - x_1} - \frac{x_2 - x_1}{4(y_D - y_1)} \right)$ .
- sinon  $y_1 \neq y_2$  et  $\lambda = \frac{1}{(y_2 - y_1)^2} \left( (x_2 - x_1)(y_D - \frac{y_1 + y_2}{2}) \pm d(P_1, P_2)\sqrt{\Delta} \right)$  et  $\Delta = (y_D - y_1)(y_D - y_2) > 0$ .

Enfin, la détection d'un événement ponctuel (resp. circulaire) entraîne l'insertion (resp. la suppression) d'un arc dans l'arbre. Ces événements sont traités ainsi :

FIGURE 19 – Insertion de l'arc  $\alpha_2$  et des breakpointes  $B_1$  et  $B_2$ .

---

#### Algorithme 4 : Algorithme pour l'insertion d'arc.

---

**Données :** Un arbre  $arb$  et un site  $P(x, y)$ .

**Résultat :** Un arbre contenant l'arc  $\alpha_2$  associé au site  $P$ .

**1 Fonction Inserer( $P$ ,  $arb$ ) :**

```

1 // Trouver l'arc au-dessus de  $P$ .
2  $\alpha_1 \leftarrow \text{RechercherFeuille}(P, arb)$ 
3  $P_1 \leftarrow \text{Site}(\alpha_1)$  // Site associé à l'arc  $\alpha_1$ .
4 // Nouveaux breakpoint
5  $B_1 \leftarrow \text{BreakPoint}(P_1, P, 'g')$ 
6  $B_2 \leftarrow \text{BreakPoint}(P_1, P, 'd')$ 
7 // Ajouter une arête au diagramme de Voronoï.
8  $\text{AjouterArête}(B_1, B_2)$ 
9  $\alpha_2 \leftarrow \text{Arc}(P)$ 
10  $\alpha_3 \leftarrow \text{Arc}(P_1)$ 
11  $\text{MettreVoisinsAJour}(\alpha_1, \alpha_2, \alpha_3, B_1, B_2)$ 
12  $N \leftarrow \text{Noeud}(B_2, \text{Noeud}(B_1, \alpha_1, \alpha_2), \alpha_3)$ 
13 // Substituer l'arc  $\alpha_1$  par le sous-arbre  $N$ .
14  $\text{Remplacer}(\alpha_1, N, arb)$ 
15  $\text{Reequilibrer}(arb)$ 

```

---

FIGURE 20 – Insertion d'un nouvel arc dans l'arbre après détection d'un événement ponctuel.

et pour ce qui est des événements circulaires :

17. Dans le cas (rare) où  $y_1 = y_2 = y_D$ , nous renvoyons  $\left( \frac{x_1 + x_2}{2}; -\infty \right)$  comme point d'intersection.

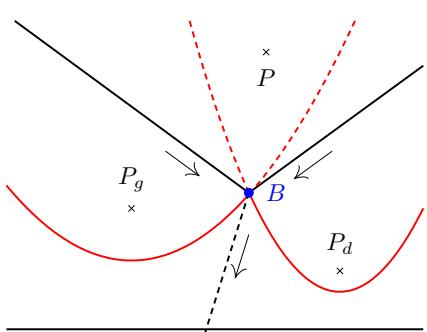


FIGURE 21 – Suppression de l’arc  $\alpha$  associé au site  $P$ .

---

**Algorithme 5 : Algorithme pour la suppression d’arc.**


---

**Données :** Un arbre  $arb$  et un arc  $\alpha$ .

**Résultat :** Un arbre ne contenant plus l’arc  $\alpha$ .

**1 Fonction** Supprimer( $\alpha$ ,  $arb$ ) :

```

1 // Trouver les noeuds de l’arbre qui bordent l’arc.
2  $N_g, N_d \leftarrow$  RechercherNoeuds( $\alpha, arb$ )
3  $N \leftarrow$  Parent( $\alpha$ ) // Parent de  $\alpha$  dans  $arb$  ( $N_g$  ou  $N_d$ )
4 // Sites des arcs voisins
5  $P_g \leftarrow$  Site(VoisinGauche( $\alpha$ ))
6  $P_d \leftarrow$  Site(VoisinDroit( $\alpha$ ))
7 // Nouveau breakpoint (choisir le bon sens...).
8  $B \leftarrow$  BreakPoint( $P_g, P_d$ )
9 MettreVoisinsAJour(VoisinGauche( $\alpha$ ), VoisinDroit( $\alpha$ ),  $B$ )
10 // Subsister le parent de  $\alpha$  par son autre fils.
    Remplacer( $\alpha$ , AutreFils( $N$ ),  $arb$ )
    // Ajouter des composantes au diagramme de Voronoï.
    AjouterAretesEtSomets( $N_g, N_d, B$ )
    // Changer l’étiquette du noeud  $N_g, N_d$  différent de  $N$ .
    ChangerEtiquette( $N_g, N_d, B$ )
    Reequilibrer( $arb$ )

```

---

FIGURE 22 – Suppression d’un arc dans l’arbre après détection d’un événement circulaire.

### 2.2.3 Modélisation du diagramme de Voronoï en construction

À mesure que les événements sont déclenchés, on maintient à jour les sommets, arêtes et faces du diagramme de Voronoï en construction. Pour cela, on utilise une structure de donnée en « demi-arêtes » (dite *half-edge* en anglais, cf. [4], chapitre 17, pour la définition) où se trouve ci-dessous une illustration (nous orientons les demi-arêtes d’une même face dans le sens anti-horaire) :

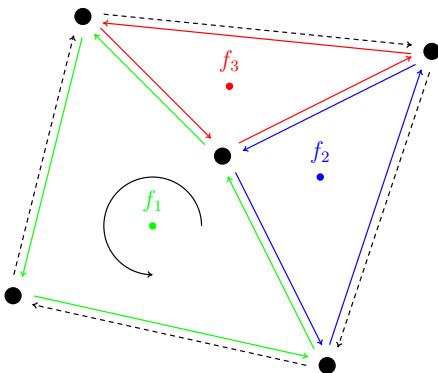


FIGURE 23 – Représentation de 3 faces  $f_1$ ,  $f_2$  et  $f_3$  avec une géométrie en demi-arête.

Voici le pseudo-code définissant les trois objets **Sommet**, **Arete** et **Face** avec leurs attributs :

---

**Algorithme 6 : Les trois objets de la géométrie en demi-arête.**


---

```

1 Classe Sommet : // Définition d’un sommet.
2   | Float x, y
3   | Tuple coordonnees
4   | Int id
5 Classe Arete : // Définition d’une demi-arête (son arête soeur est l’autre moitié de l’arête).
6   | Arete arete_soeur, arete_suivante, arete precedente
7   | Sommet sommet_but
8   | Face face
9   | Int id
10 Classe Face : // Définition d’une face.
11   | Arete arete_associee
12   | Int id

```

---

**Remarque :** La complexité spatiale de cette structure est donnée est linéaire en le nombre de composantes du diagramme de Voronoï donc, linéaire en le nombre  $n$  de sites.

De plus, comme chaque arête du diagramme de Voronoï est tracée par un unique couple de breakpoints<sup>18</sup> (nous qualifierons les breakpoints d'un tel couple de « jumeaux ») nous imposons l'invariant :

**Invariant :** Chaque demi-arête est tracée par un unique breakpoint et, des breakpoints jumeaux tracent des arêtes sœurs.

### 3 Ajustement du tracé des zones aux routes

Le diagramme de Voronoï construit précédemment ne tient pas compte de la géographie du centre-ville (cf. introduction, 3)). Pour ajuster les arêtes du diagramme de Voronoï aux routes, on procède en deux temps :

- 1) construire le graphe des routes des centre-ville,
- 2) pour chaque sommet du diagramme de Voronoï, le substituer par le sommet du graphe routier le plus proche puis, redéfinir l'arête via une recherche de plus court chemin entre ses deux extrémités.

#### 3.1 Construction du graphe des routes

Nous utilisons le site *Open Street Map* qui fournit un fichier (au format *xml*) de données géographiques du centre-ville<sup>19</sup>. Le fichier obtenu contient principalement des noeuds et des routes :

```
<node id="2831" lat="44.846" lon="-0.570"/>
```

(a) Un noeud

```
<way id="79402510">
  <nd ref="926763722"/>
  [...]
  <nd ref="8531411025"/>
  <tag k="highway" v="service"/>
</way>
```

(b) Une route : succession ordonnée de noeuds

Nous extrayons de ce fichier seulement les routes pertinentes ainsi que leurs noeuds. Nous représentons ces données :

- avec dictionnaire de la forme {id du noeud : longitude/latitude} pour les noeuds,
  - avec une liste de chaînes (les identifiants des noeuds) pour les routes
- puis, nous les mémorisons dans deux fichiers : *noeuds.json* et *routes.json*.

Enfin, nous imposons au graphe des routes d'être non orienté et pondéré : chaque arête se voit attribuer la distance réelle (en mètre) qui sépare ses deux extrémités sur la carte. Nous représentons ce graphe par un dictionnaire de la forme {id du sommet : dictionnaire des voisins} : {"15811": {"20013" : 16.49872, [...], "16818" : 24.41298}, [...]}

Voici le graphe des routes une fois affiché sur (une partie) de la carte :

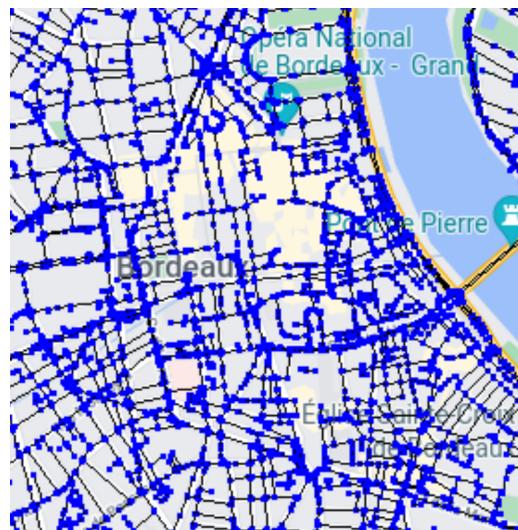


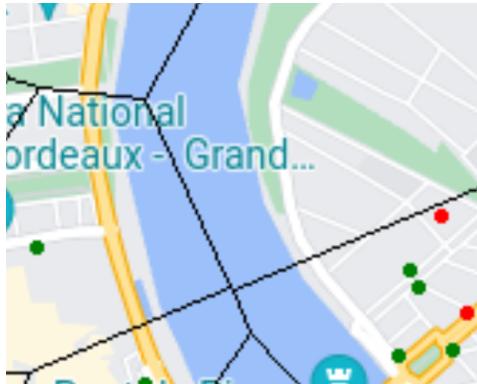
FIGURE 25 – Le graphe des routes : en bleu les sommets et en noir, les arêtes.

18. Ou bien les deux breakpoints se meuvent le long d'une arête, en des sens opposés ou bien, l'un est immobile et l'autre trace toute l'arête seul (ce dernier cas se produit exclusivement pour les breakpoints issus de la convergence de deux breakpoints antérieurs).

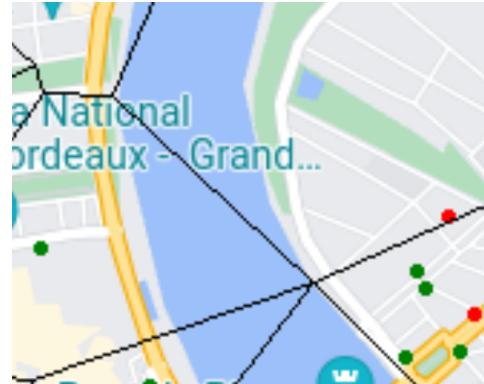
19. L'extraction de données depuis une région délimitées par l'utilisateur est possible, à l'adresse *Open Street Map - centre-ville de Bordeaux* se trouve la zone dont nous avons exploité les données.

### 3.2 Ajustement des arêtes du diagramme de Voronoï aux routes

Afin d'ajuster les sommets du diagramme de Voronoï aux sommets du graphe, nous réutilisons les arbres 2-D pour effectuer une recherche d'un plus proche voisin. Voici ce que nous obtenons avant, et après cette recherche :



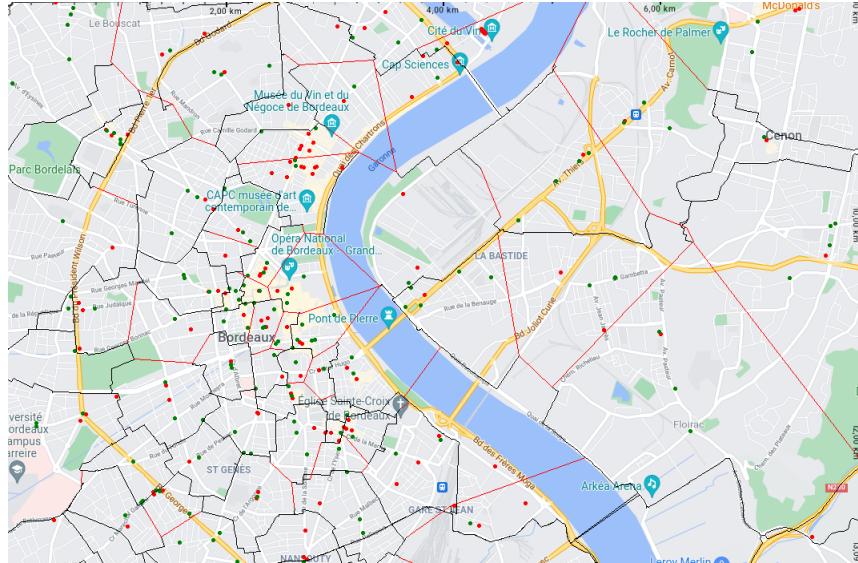
(a) Avant ajustement des sommets.



(b) Après ajustement des sommets.

Puis, nous devons nous assurer que le graphe des routes est connexe. Pour cela, on extrait l'ensemble de ses composantes connexes en ne conservant que celle avec le plus grand nombre de sommets (environ 46000).

Enfin, pour ajuster les arêtes du diagramme de Voronoï aux routes, nous effectuons la recherche de plus court chemin avec l'algorithme de Dijkstra.<sup>20</sup> Nous obtenons finalement :



**Remarque :** En noires sont les arêtes correctement ajustées aux routes, en rouges celles dont un plus court chemin excède de 1,5 fois la distance à vol d'oiseau.

## 4 Conclusion

Nous sommes parvenus à construire un découpage du centre-ville de Bordeaux ; ce découpage vérifie convenablement les trois critères que nous nous étions fixés (cf. introduction), malgré la présence de cellules vides (plus de magasins) ou des arêtes qui se croisent parfois.

Notre implémentation de l'algorithme de Fortune a une complexité temporelle en  $O(n \log(n))$  pour  $n$  sites. En effet, le nombre d'événements déclenchés est un  $O(n)$ , et chacun est traité en un  $O(\log(n))$  dans le pire cas. Soit  $s$  (resp.  $a$ ) le nombre de sommets (resp. arêtes) du graphe des routes alors, l'extractin des composantes connexes est un  $O(s + a)$  et l'algorithme de Dijkstra en  $O((s + a) \log(s))$ . Le nombre d'arêtes du diagramme de Voronoï étant un  $O(n)$  nous en tirois une complexité en  $O(n(s + a) \log(s))$  pour la construction d'un tel découpage.

Toutefois, nous tenons à souligner que notre algorithme peut échouer selon la configuration des sites, notamment lors de la suppression d'un arc  $\alpha$ . En effet, cet arc  $\alpha$  étant réduit à un point, les breakpoints qui bordent  $\alpha$  sont confondus ce qui rend la recherche de  $\alpha$  dans l'arbre équilibré délicate. Nous avons opté pour relever légèrement la droite de balayage de  $10^{-20}$  (quantité arbitraire), mais des solutions plus robuste sont envisageables : remplacer l'arbre par une liste chaînée (mais cela à un coût puisque la recherche dans une telle structure n'est plus un  $O(\log(n))$ ).

20. L'algorithme de Floyd-Warshall n'est pas utilisable en pratique, le nombre de sommets du graphe des routes est trop important, conduisant, de part sa complexité temporelle en  $O(s^3)$ , à un nombre d'opérations de l'ordre de  $46000^3 \approx 10^{14}$ .

## 5 Annexes

### Code pour l'extraction des magasins

Ci-dessous le code pour l'extraction des magasins depuis la base de donnée fournie par le site *Ça reste ouvert* :

```

1 import csv
2
3 def ecrire_csv(nom, lignes):
4     '''Écrire un fichier CSV ligne à ligne.'''
5     with open(nom, mode = 'w+', newline = '', encoding = 'utf-8') as fichier:
6         writer = csv.writer(fichier, delimiter = ',')
7         for l in lignes:
8             writer.writerow(l)
9
10 # Structure du fichier csv initial,
11 # Colonnes : osm_id, name, category, subcategory, brand, wikidata, url_hours, infos, status, opening_hours, lat, lon.
12 def extraire_boutiques():
13     # Les catégories des boutiques conservées.
14     alimentation_categories = ['greengrocer', 'grocery', 'marketplace', 'supermarket']
15     sante_categories = ['medical_supply', 'pharmacy', 'health_center', 'optician']
16
17     # Liste des boutiques suivant qu'il s'agit d'une boutique alimentaire ou médicale.
18     alimentation = []
19     sante = []
20
21     # Extraction des boutiques pertinentes à l'échelle nationale.
22     with open('../Data/BDD_ca_reste_ouvert.csv', encoding = 'utf-8') as fichier:
23         for ligne in csv.reader(fichier, delimiter = ','):
24             latitude, longitude = float(ligne[-2]), float(ligne[-1])
25             cat, subcat = ligne[2], ligne[3]
26             if cat == 'food' and subcat in alimentation_categories:
27                 alimentation.append(ligne)
28             elif cat == 'health' and subcat in sante_categories:
29                 sante.append(ligne)
30
31     # Sauvegarde.
32     ecrire_csv("../Data/Alimentation.csv", alimentation)
33     ecrire_csv("../Data/Sante.csv", sante)
```

Et voici le code qui permet de charger en mémoire les supermarchés et pharmacies de la région d'étude seulement :

```

1 # Pour la manipulation d'images.
2 from PIL import Image, ImageDraw
3
4 #####
5 # Constantes (accès global) :
6 image = "../Data/CentreVille.png"
7 carte = Image.open(image)
8 draw = ImageDraw.Draw(carte, 'RGBA') # Pour la transparence.
9
10 longueur, largeur = carte.size # Longueur et largeur de la carte.
11
12 # Latitude et longitude des coins supérieurs gauche & inférieur droit et les ratios associés (pour la mise à l'échelle).
13 longitude_min, longitude_max = -0.60696, -0.50681
14 longitude_ratio = longueur / (longitude_max - longitude_min)
15
16 latitude_min, latitude_max = 44.81924, 44.86563
17 latitude_ratio = largeur / (latitude_max - latitude_min)
18 #####
19
20 def polaire2pixel(point):
21     '''Renvoie les coordonnées du pixel correspondant aux coordonnées polaires.'''
22     lat, lon = point
23     x, y = int((lon - longitude_min) * longitude_ratio), int((latitude_max - lat) * latitude_ratio)
24     return (x, y)
25
26 def estDansRectangle(p, c = (0, 0), L = longueur, l = largeur):
27     '''Tester si le point p est dans le rectangle de coin supérieur gauche c, de longueur L et de largeur l.'''
28     return c[0] <= p[0] < c[0] + L and c[1] <= p[1] < c[1] + l
29
30 def construirePoints(fichier, L = 10, l = 11):
31     '''L : indice de la longitude, l : indice de la latitude.'''
32     tableau = []
33     with open(fichier, encoding = 'utf-8') as csvfile:
```

```

34     for ligne in csv.reader(csvfile, delimiter = ','):
35         longitude, latitude = float(ligne[0]), float(ligne[1])
36         x, y = polaire2pixel((latitude, longitude))
37
38         if estDansRectangle((x, y)): # Vérifier si le magasin est dans la zone d'étude.
39             tableau.append((x, y))
40
41     return tableau
42
43 def dessinerPoints(points, couleur, rayon = 1, canvas = draw):
44     '''Dessiner sur la carte des points selon leur coordonnées (x, y).'''
45     for (x, y) in points:
46         canvas.ellipse((x - rayon, y - rayon, x + rayon, y + rayon), fill = couleur, outline = couleur)
47
48 # Construction du tableau des supermarchés (resp. pharmacies) du centre-ville de Bordeaux.
49 alimentaire = construirePoints('../Data/Alimentation.csv')
sante = construirePoints('..../Data/Sante.csv')

```

## Arbres 2-D : preuves et codes

*Preuve :* (de la propriété 1) La terminaison de l'algorithme est assurée par le fait qu'à chaque appel récursif, l'écart  $|M - m|$  entre les deux indices du tableau diminue, formant une suite strictement décroissante d'entiers naturels.

Le second point<sup>21</sup> s'établit par récurrence forte sur  $n$ , on pose  $H(k)$  : « pour tout tableau de  $k$  point deux-à-deux distincts du plan, l'algorithme 1 renvoie un arbre binaire équilibré de hauteur  $1 + \lfloor \log_2(k) \rfloor$  ». Pour  $k = 1$ , la propriété est vrai (car l'arbre est de hauteur 1, donc équilibré), et l'est encore pour  $k = 2$ . Supposons la propriété vraie pour tout  $k \in \llbracket 1; n \rrbracket$  avec  $n \geq 2$  alors, étant donné  $n + 1 \geq 3$  points distincts du plan :

- si  $n + 1$  est pair i.e.  $n + 1 = 2p$  avec  $p \geq 2$  alors, d'après  $H\left(\frac{n+1}{2} - 1\right) = H(p - 1)$  (resp. d'après  $H(p)$ ) le fils gauche (resp. droit) de la racine de l'arbre renvoyé est équilibré, de hauteur  $1 + \lfloor \log_2(p - 1) \rfloor$  (resp.  $1 + \lfloor \log_2(p) \rfloor$ ). De plus,  $0 \leq (1 + \lfloor \log_2(p) \rfloor) - (1 + \lfloor \log_2(p - 1) \rfloor) \leq 1$  donc, l'arbre renvoyé est équilibré, de hauteur  $1 + (1 + \lfloor \log_2(p) \rfloor) = 1 + \lfloor \log_2(2p) \rfloor$ .
- si  $n + 1$  est impair,  $n + 1 = 2p + 1$  avec  $p \in \mathbb{N}^*$  et d'après  $H(p)$ , les fils gauche et droit de la racine de l'arbre renvoyé sont équilibrés, de même hauteur  $1 + \lfloor \log_2(p) \rfloor$  : cet arbre est donc équilibré, de hauteur  $2 + \lfloor \log_2(p) \rfloor$  et, si  $2^k \leq p < 2^{k+1}$  pour un certain  $k \in \mathbb{N}^*$  alors  $2^k < p + \frac{1}{2} < 2^{k+1}$  d'où  $2 + \lfloor \log_2(p) \rfloor = 2 + \lfloor \log_2(p + \frac{1}{2}) \rfloor = 1 + \lfloor \log_2(2p + 1) \rfloor$ .

Ceci achève la preuve.

*Preuve :* (de la propriété 2) L'arbre binaire renvoyé contient  $n$  noeuds de taille  $O(1)$  donc, la complexité spatiale de l'algorithme 1 est un  $O(n)$ . Pour la complexité temporelle, notons  $C(n)$  le nombre d'opérations effectuées par l'algorithme 1 pour construire l'arbre avec un tableau de  $n$  points du plan, il en ressort que,  $C(0) = 0$ ,  $C(1) = 1$  et, qu'il existe une constante  $\alpha > 0$  telle que, pour tout  $n \in \mathbb{N}^*$  :

$$C(n+1) \leq \underbrace{\alpha n \log_2(n)}_{\text{tri fusion sur le tableau}} + \underbrace{C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{construction du fils gauche}} + \underbrace{C\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{construction du fils droit}}$$

On pose  $(D_n)_{n \in \mathbb{N}}$  la suite définie par :  $D_0 = 0$ ,  $D_1 = 1$  et pour tout  $n \in \mathbb{N}^*$ ,  $D_{n+1} = \alpha n \log_2(n) + D_{\lfloor \frac{n}{2} \rfloor} + D_{\lceil \frac{n}{2} \rceil}$ . On montre (nous ne le faisons pas) que la suite  $(D_n)_{n \in \mathbb{N}}$  est croissante et vérifie  $C(n) \leq D_n$  pour tout  $n \in \mathbb{N}$ . Il vient, pour  $n \geq 1$  qu'il existe  $p \in \mathbb{N}^*$  tel que  $2^{p-1} \leq n < 2^p$  d'où :

$$\begin{aligned} D_n &\leq D_{2^p} = \alpha p 2^p + 2D_{2^{p-1}} && \text{puis, par croissance des } (D_n)_{n \in \mathbb{N}} \\ &= \alpha 2^p \times \left( \sum_{k=1}^p k \right) + 2^p D_1 && \text{en raisonnant de proche en proche} \\ &= \alpha p(p+1)2^{p-1} + 2^p \\ &= O(p^2 \times 2^p) \\ &= O(n \log^2(n)) \end{aligned}$$

d'où la complexité souhaitée.

*Preuve :* (de la propriété 3) Montrons, par récurrence sur le nombre  $n \in \mathbb{N}^*$  de points, que l'algorithme 2 termine et est correct, soit  $H(n)$  : « pour tout arbre 2-D à  $n$  noeuds, l'algorithme 2 termine et renvoie un plus proche voisin (parmi les noeuds de l'arbre) du point  $P$  (fixé) ». Pour  $n = 1$ , l'algorithme termine (il n'y a qu'une feuille) et renvoie bien le noeud de cette feuille :  $H(1)$  est vraie. Supposons la propriété  $H$  vraie pour tout  $k \in \llbracket 1; n \rrbracket$  et considérons un arbre 2-D avec  $n+1$  noeuds. La racine n'étant pas une feuille, l'algorithme va descendre dans l'arbre, quitte à supposer que l'on descende dans le fils gauche,

21. Un arbre binaire équilibré est un arbre binaire tel que pour tout noeud, son balancement (i.e. la différence entre les hauteurs de ses fils droits et gauches) est dans  $\{-1, 0, 1\}$ .

- si le fils gauche est vide, on remonte à la racine. Le plus proche voisin est alors mis à jour puis, s'il y a basculement dans le fils droit (celui-ci est alors non vide), l'hypothèse de récurrence montre qu'en un nombre fini d'étapes on remonte à la racine de l'arbre. D'ailleurs, on renvoie bien un plus proche voisin de  $P$  car il ne peut s'agir que de la racine de l'arbre ou, d'un noeud du fils droit (qui est correctement détecté par hypothèse de récurrence).
- sinon, en appliquant l'hypothèse de récurrence, la terminaison implique que l'on remonte à la racine et, on connaît un plus proche voisin de  $P$  parmi les noeuds du fils gauche. On vérifie alors si la racine est plus proche ou non de  $P$  puis, on opère le basculement comme au cas précédent. Au final, l'algorithme termine et renvoie un plus proche voisin de  $P$  (qui a bien été détecté).

Finalement, on l'algorithme termine et renvoie un plus proche voisin de  $P$ , ceci achève la récurrence.

Voici les codes concernant les arbres 2-D :

```

1 # Tri fusion :
2 def fusionner(t, g, m, d, comp):
3     '''Fusion ordonnée de t[g:m] et t[m:d] dans t[g:d].'''
4     i, j = 0, m - g
5     t2 = t[g:d] # On mémorise les cases qui vont être modifiées.
6     for k in range(g, d):
7         if i < m - g and (j == d - g or comp(t2[i], t2[j])):
8             t[k] = t2[i]
9             i += 1
10        else:
11            t[k] = t2[j]
12            j += 1
13
14 def tri_fusion(t, m = 0, M = None, comp = lambda x, y : x < y):
15     '''Tri de t[m:M] par tri fusion sur place.'''
16     def tri_aux(g, d):
17         '''Trier le sous-tableau t[g:d].'''
18         if g < d - 1: # Au moins deux éléments.
19             m = (g + d) // 2
20             tri_aux(g, m)
21             tri_aux(m, d)
22             fusionner(t, g, m, d, comp) # À l'issu, t[g:d] sera trié.
23         if M == None: M = len(t)
24     tri_aux(m, M)
25 #####
26
27 def dist2(p1, p2 = (0, 0)):
28     '''Norme L2 au carré, par défaut (si p2 n'est pas fourni) on renvoie la norme au carré.'''
29     (x1, y1), (x2, y2) = p1, p2
30     return (x1 - x2)**2 + (y1 - y2)**2

```

Pour l'arbre 2-D, on le représente par une classe :

```

1 class Arbre2D:
2     class Noeud:
3         def __init__(self, etiquette, filsG = None, filsD = None):
4             '''Noeud (noeud interne ou feuille).'''
5             self.etiquette = etiquette
6             self.filsG, self.filsD = filsG, filsD
7
8         def estFeuille(self):
9             '''Tester si un noeud est une feuille.'''
10            return self.filsG == self.filsD == None
11
12    def __init__(self, points):
13        '''Arbre (binaire) 2-d.'''
14        self.points = points
15
16    def construire(self, m, M, axe):
17        '''Construit récursivement un arbre 2-d.'''
18        def comp_coordonnee(p1, p2):
19            '''Fonction pour comparer deux points suivant leur coordonnée en x ou en y.'''
20            if axe == 1: return p1 < p2
21            return p1[::-1] < p2[::-1] # On renverse les couples pour les comparer.
22
23        if m >= M:
24            return None
25        milieu = (m + M) // 2
26        return self.Noeud(self.points[milieu], self.construire(m, milieu, 1 - axe), self.construire(milieu + 1, M, 1 - axe))

```

```

28
29     def plusProcheVoisin(self, arb, point, axe):
30         '''Recherche du plus proche voisin parmi les points de l'arbre.'''
31         meilleur_point, meilleure_distance = None, float('inf')
32
33     def mettre_a_jour(p1):
34         '''Mettre à jour, si nécessaire, le meilleur point.'''
35         nonlocal meilleure_distance, meilleur_point
36         d = dist2(p1, point)
37         if d < meilleure_distance:
38             meilleur_point, meilleure_distance = p1, d
39
40     def basculer(noeud, fils, direction):
41         '''Tester s'il faut parcourir ("basculer" dans) le fils du nud.'''
42         if fils == None:
43             return False
44         d1, d2 = abs(point[direction] - noeud.etiquette[direction]), abs(point[1 - direction] - fils.etiquette[1 -
45             direction])
46         return (min(d1, d2) < meilleure_distance) or (d1**2 + d2**2 < meilleure_distance)
47
48     def recherche(arbre, direction):
49         # Cas de base (arbre vide ou feuille).
50         if arbre == None:
51             return None
52         elif arbre.estFeuille(): # On met à jour le plus proche voisin trouvé puis, on remonte.
53             mettre_a_jour(arbre.etiquette)
54             return None
55         # Descente dans l'arbre.
56         if point[direction] < arbre.etiquette[direction]: # Descente à gauche.
57             recherche(arbre.filsG, 1 - direction)
58             mettre_a_jour(arbre.etiquette)
59             if basculer(arbre, arbre.filsD, direction): # Pour basculer dans l'autre fils si besoin.
60                 recherche(arbre.filsD, 1 - direction)
61             else: # Descente à droite.
62                 recherche(arbre.filsD, 1 - direction)
63                 mettre_a_jour(arbre.etiquette)
64                 if basculer(arbre, arbre.filsG, direction): # Pour basculer dans l'autre fils si besoin.
65                     recherche(arbre.filsG, 1 - direction)
66         recherche(arbre, axe)
67         return meilleur_point

```

## Construction des groupes de points

Il ne reste plus qu'à construire les groupes de magasins et afficher leur enveloppe convexe :

```

1     def regrouper(pharmacies, supermarches):
2         '''Regrouper supermarche à la pharmacie la plus proche (on suppose p1 et p2 non vides).'''
3         grp = {p: [p] for p in pharmacies} # Groupe associé aux pharmacies.
4         voisin_supermarche = {p: None for p in supermarches} # Chaque supermarché est associé à la pharmacie la plus proche.
5
6         # Construction des deux arbres 2-D :
7         arb_p, arb_s = Arbre2D(pharmacies), Arbre2D(supermarches)
8         racine_p, racine_s = arb_p.construire(0, len(pharmacies), 0), arb_s.construire(0, len(supermarches), 0)
9
10        # On associe chaque supermarché à la pharmacie la plus proche.
11        for p in supermarches:
12            voisin = arb_p.plusProcheVoisin(racine_p, p, 0)
13            grp[voisin].append(p)
14            voisin_supermarche[p] = voisin
15
16        # Pour les pharmacies dont leur groupe est réduit à elle-même, on les met dans
17        # le même groupe que celui du supermarché le plus proche.
18        for (p, l) in grp.items():
19            if len(l) == 1: # Pharmacie seule dans son groupe.
20                voisin = arb_s.plusProcheVoisin(racine_s, p, 0)
21                pharmacie_associee = voisin_supermarche[voisin]
22                grp[pharmacie_associee].append(p)
23
24        # On ne renvoie que les groupes contenant au moins deux magasins (i.e. au moins un
25        # supermarché et une pharmacie) :
26        return [e for e in grp.values() if len(e) > 1]

```

Puis, on affiche ces groupes avec leur enveloppe convexe :

```

1 from mpmath import *
2 mp.dps = 30 # On demande une précision à 30 chiffres après la virgule sur les flottants (cf. Voronoï).
3
4 ######
5 # Quelques fonctions utiles :
6 def vecteur(p, q):
7     '''Renvoie le vecteur associé au bipoint pq.'''
8     return [qi - pi for (pi, qi) in zip(p, q)]
9
10 def determinantVect2(p1, p2, p3):
11     '''Calcul le déterminant du système de vecteurs plans (p1p2, p1p3)'''
12     return det([vecteur(p1, p2), vecteur(p1, p3)])
13 #####
14
15 # Algorithme d'Andrew :
16 def enveloppe_convexe(P):
17     '''Renvoie l'enveloppe convexe de P.'''
18     tri_fusion(P) # L'ordre lexicographique est naturellement utilisée sur les tuples/listes en Python.
19     L = []
20     # Calcul de l'enveloppe convexe supérieure.
21     for p in P:
22         while len(L) >= 2 and determinantVect2(L[-2], L[-1], p) <= 0: L.pop()
23         L.append(p)
24     l = len(L)
25     # Calcul de l'enveloppe convexe inférieure.
26     for i in range(len(P) - 2, -1, -1):
27         while len(L) >= l + 1 and determinantVect2(L[-2], L[-1], P[i]) <= 0: L.pop()
28         L.append(P[i])
29     return L # Le dernier point est en double (c'est pour le dessin).
30
31 def dessiner_enveloppe_convexe(env, canvas = draw):
32     '''Dessine les enveloppes convexes des groupes sur la carte.'''
33     for grp in env:
34         for i in range(len(grp) - 1):
35             canvas.line((grp[i], grp[i + 1]), fill = 'green')
36 #####
37
38 grp = regrouper(sante, alimentaire) # Constitue les groupes.
39 evn = list(map(enveloppe_convexe, grp)) # Construire les enveloppes convexes.
40
41 # Dessiner des enveloppes convexes.
42 dessiner_enveloppe_convexe(evn)
43
44 # Dessiner les magasins :
45 dessinerPoints(alimentaire, 'red', 2)
46 dessinerPoints(sante, 'green', 2)

```

## Diagrammes de Voronoï : preuves et codes de l'algorithme de Fortune

*Preuve :* (de la propriété 6) Soit  $Q \in \mathbb{R}^2$ , pour le premier point :

- si  $Q$  est sur une arête du diagramme de Voronoï alors, par définition, il existe deux sites distincts  $P_i$  et  $P_j$  tels que  $r = d(Q, P_i) = d(Q, P_j)$  donc  $Q \in C_i \cap C_j$ . De plus, par l'absurde, s'il existait un  $m \in \llbracket 1; n \rrbracket$  tel que  $d(Q, P_m) < r$  alors  $Q \notin C_i$  puisqu'il est plus proche du site  $P_m$  que de  $P_i$ , ce qui ne se peut. Ainsi, pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $d(Q, P_m) \geq r$ .
- réciproquement, si  $Q$  est à équidistance de deux sites distincts  $P_i$  et  $P_j$  et si, pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $d(Q, P_m) \geq r = d(Q, P_i)$  alors, par définition,  $Q \in C_i \cap C_j$  donc, est sur la médiatrice de  $[P_i P_j]$  soit,  $Q$  est sur une arête du diagramme de Voronoï.

Pour le second point :

- si  $Q$  est un sommet du diagramme de Voronoï, par définition, il existe trois sites distincts  $P_i$ ,  $P_j$  et  $P_k$  tels que  $r = d(Q, P_i) = d(Q, P_j) = d(Q, P_k)$ . Or,  $Q$  étant le point d'intersection de deux arêtes du diagramme de Voronoï (par exemple, les arêtes associées aux couples  $(P_i, P_j)$  et  $(P_j, P_k)$ ) alors, d'après ce qui précède, pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $d(Q, P_m) \geq r$ .
- réciproquement, si  $Q$  est à équidistance de trois sites distincts  $P_i$ ,  $P_j$  et  $P_k$  et si, pour tout  $m \in \llbracket 1; n \rrbracket$ ,  $d(Q, P_m) \geq r = d(Q, P_i)$  alors,  $Q \in C_i \cap C_j \cap C_k$ . En particulier, d'après ce qui précède,  $Q$  est sur l'intersection des médiatrices de  $[P_i P_j]$  et de  $[P_j P_k]$  donc, c'est un sommet du diagramme de Voronoï puisque  $Q$  est à l'intersection de deux arêtes de ce diagramme.

Ceci conclut la preuve.

Voici le code concernant l'implémentation d'une file de priorité par un tas binaire min :

```

1 # Implémentation d'un tas (binaire) min (pour une file de priorité).
2 class TasMin:
3     def __init__(self, n, l = [], comp = lambda x, y: x < y, indexation = False):
4         '''Tas binaire max à n éléments.'''
5         self.t = [None] * n # Taille du tas-min.
6         self.idx = 0 # Indice de la position "libre".
7         self.comp = comp # Fonction de comparaison.
8         # Pour mémoriser l'indice des objets qui se trouve dans le tableau (facilite leur traitement, sans parcourir le
9         # tas).
10        self.indexation = indexation
11
12    for e in l:
13        self.ajouter(e) # Si des éléments sont passés en argument, on les ajoute dans le tas.
14
15    def estPlein(self): return self.idx == len(self.t) # Tester si le tas est plein.
16    def estVide(self): return self.idx == 0 # Tester si le tas est vide.
17
18    def parent(self, i): return (i - 1) // 2 # Renvoie l'indice du nud parent.
19    def filsG(self, i): return 2 * i + 1 # Renvoie l'indice du fils gauche.
20    def filsD(self, i): return 2 * i + 2 # Renvoie l'indice du fils droit.
21
22    def echanger(self, i, j):
23        '''Échanger deux éléments dans un tableau.'''
24        self.t[i], self.t[j] = self.t[j], self.t[i]
25        if self.indexation: # Si il y a indexation sur les éléments du tas, on met à jour les indices.
26            if self.t[i] != None: self.t[i].idx = i
27            if self.t[j] != None: self.t[j].idx = j
28
29    def percole_haut(self, i):
30        '''Faire "remonter" la clé d'indice i si besoin.'''
31        j = self.parent(i)
32        while 0 <= j and self.comp(self.t[i], self.t[j]): # comp(x, y), si x < y
33            self.echanger(i, j)
34            i, j = j, self.parent(j)
35
36    def percole_bas(self, i):
37        '''Faire "descendre" la clé d'indice i si besoin.'''
38        g, d = self.filsG(i), self.filsD(i)
39        # Condition : si les deux fils sont définis et si l'invariant du tas n'est pas satisfait.
40        while d < self.idx and (self.comp(self.t[g], self.t[i]) or self.comp(self.t[d], self.t[i])):
41            m = g if self.comp(self.t[g], self.t[d]) else d
42            self.echanger(i, m)
43            g, d, i = self.filsG(m), self.filsD(m), m
44
45        # Si le fils droit n'est pas défini mais, que le fils gauche l'est et que l'invariant du tas n'est pas vérifié.
46        if d == self.idx and self.comp(self.t[g], self.t[i]):
47            self.echanger(i, g)
48
49    def ajouter(self, e):
50        '''Ajouter une clé.'''
51        if self.estPlein(): raise Exception('Tas plein !')
52        self.t[self.idx] = e
53        if self.indexation: e.idx = self.idx
54        self.percole_haut(self.idx)
55        self.idx += 1
56
57    def extraire(self, i):
58        '''Extraire la clé d'indice i.'''
59        if self.estVide(): raise Exception('Tas vide !')
60        v, self.t[i] = self.t[i], None
61        self.idx -= 1
62        self.echanger(i, self.idx)
63        self.percole_bas(i)
64        return v
65
66    def minimum(self):
67        return self.extraire(0)

```

Ci-dessous se trouve le code permettant le calcul des coordonnées des breakpoints, des sommets du diagramme de Voronoï et de vérifier si deux demi-droites de coupent :

```

1 def isobarycentre(points):
2     '''Renvoie les coordonnées de l'isobarycentre des points du plan fournis.
3     Hypothèse : la liste des points est non vide !'''
4     X, Y, n = mpf(0), mpf(0), len(points)

```

```

5   for (x, y) in points:
6     X += mpf(x)
7     Y += mpf(y)
8   return [X / n, Y / n]
9
10 def positivementLie(v1, v2, eps = 1e-20):
11   '''Tester si deux vecteurs du plan son positivement liés, une précision est fournie.'''
12   # Si v2 est nul, 0 * v1 = v2, sinon si v1 et v2 sont colinéaires avec un produit scalaire positif on renvoie True.
13   return v2 == [0, 0] or (abs(det([v1, v2])) <= eps and fdot(v1, v2) >= 0)
14 #####
15
16 def intersectionDroites(p1, v1, p2, v2):
17   '''Renvoie l'unique point d'intersection entre les deux droites d1 et d2 passant par p1 (resp. p2) de vecteur
18   directeur v1 (resp. v2). Aussi, les droites ne doivent pas être parallèles (ni confondues).'''
19   assert v1 != [0, 0] and v2 != [0, 0], 'Vecteur directeur nul !' # Ce test peut être redondant...
20   assert det([v1, v2]) != 0, 'Droites parallèles.' # Idem...
21   # On se ramène à la résolution d'un système linéaire 2x2.
22   # On cherche x, y tels que p1 + x*v1 = p2 + y*v2 i.e. p2 - p1 = x*v1 - y*v2 = (v1 v2) * (x - y) (matrice de v1, v2).
23   mat = matrix([[v1[0], v2[0]], [v1[1], v2[1]]])
24   vec = vecteur(p1, p2)
25   x, ny = (mat**(-1)) * matrix([[vec[0]], [vec[1]]]) # Produit entre une matrice et un vecteur.
26   return matrix([[p1[0]], [p1[1]]]) + x * matrix([[v1[0]], [v1[1]]]) # Le point d'intersection.
27
28 def appartientDemiDroite(p, p1, v1):
29   '''Tester si le point p appartient à la demi-droite (p1, v1).'''
30   assert v1 != [0, 0], 'Pas de demi-droite !'
31   # p est sur la demi-droite d'origine p1 et de vecteur directeur v1ssi il existe x >=0 tel que p = p1 + x*v1
32   # i.e. les vecteurs p - p1 et v1 sont positivement liés.
33   return positivementLie(vecteur(p1, p), v1)
34
35 def intersecteDemiDroite(p1, v1, p2, v2):
36   '''Pour deux demi-droites, renvoie True si les demi-droites passées en argument s'intersectent dans le plan.
37   On a p1, p2 deux points du plan et v1, v2 les vecteurs directeurs de ces demi-droites respectives.'''
38   assert v1 != [0, 0] and v2 != [0, 0], 'Vecteur directeur nul !'
39   if p1 == p2: return True # Même origine.
40   if det([v1, v2]) == 0: # Les vecteurs v1 et v2 son colinéaires (rare...).
41     # On vérifie si : les droites associées aux demi-droites sont confondues et si
42     # l'origine de l'une d'elle appartient à l'autre demi-droite.
43     return det([v1, vecteur(p1, p2)]) == 0 and (appartientDemiDroite(p1, p2, v2) or appartientDemiDroite(p2, p1, v1))
44   # Sinon, les droites associées à ces demi-droites ont un unique point d'intersection
45   # dont on teste son appartenance aux demi-droites.
46   p = intersectionDroites(p1, v1, p2, v2)
47   return appartientDemiDroite(p, p1, v1) and appartientDemiDroite(p, p2, v2)
48
49 # Centre du cercle circonscrit, intersection de deux paraboles et de deux droites.
50 def cercleCirconscrit(p1, p2, p3):
51   '''Renvoie le rayon et le centre du cercle circonscrit aux points p1, p2 et p3 (s'il ne sont pas colinéaires).'''
52   x, y = intersectionDroites(isobarycentre([p1, p2]), ortho(vecteur(p1, p2)), isobarycentre([p1, p3]),
53   ↪ ortho(vecteur(p1, p3)))
54   return (x, y), sqrt(dist2(p1, (x, y)))
55
56 def intersectionParaboles(p1, p2, y, sens):
57   '''Pour deux points, renvoie le point d'intersection (à gauche 'g' ou à droite 'd' suivant le sens) entre les deux
58   ↪ paraboles
59   de foyers p1, p2 et de directrice la droite horizontale d'ordonnée y.'''
60   if p1[1] == p2[1]: # Dans ce cas leur deux breakpoints (p1, p2) et (p2, p1) seront confondues.
61     assert p1[0] != p2[0], 'Mêmes foyers !'
62     if p1[1] == y: # Cas particulier (rare...).
63       return ((p1[0] + p2[0]) / 2, float('-inf'))
64     lbd = ((p1[1] - y) / (p2[0] - p1[0]) - (p2[0] - p1[0]) / (4 * (p1[1] - y))) / 2
65   else:
66     # IMPORTANT : renvoie le point d'intersection à GAUCHEssi sens = 'g' sinon, c'est le point à DROITE.
67     e = -1
68     if (p1[1] > p2[1] and sens == 'g') or (p1[1] < p2[1] and sens == 'd'):
69       e = 1
70     Delta = (y - p1[1]) * (y - p2[1]) * dist2(p1, p2)
71     assert Delta >= 0, 'Intersection de paraboles vide !'
72     lbd = ((p2[0] - p1[0]) * (y - (p1[1] + p2[1]) / 2) + e * sqrt(Delta)) / (p2[1] - p1[1])**2
73
74   return (lbd * (p2[1] - p1[1]) + (p1[0] + p2[0]) / 2, lbd * (p1[0] - p2[0]) + (p1[1] + p2[1]) / 2)

```

Voici le code définissant la géométrie :

```

1 # Implémentation d'une géométrie dite en "demi-arêtes".
2 # Complexité (spatiale) : linéaire en le nombre de composantes de la géométrie.
3 ID_SOMMET = 0 # Identifiant pour les sommets.

```

```

4 ID_ARETES = 0 # Identifiant pour les arêtes.
5 ID_FACE = 0 # Identifiant pour les faces.
6
7 # Liste des composantes du diagrammes de Voronoi :
8 liste_sommets = []
9 liste_aretes = []
10 liste_faces = []
11
12 class Sommet:
13     def __init__(self, x, y):
14         global ID_SOMMET
15         self.x, self.y = x, y
16
17         self.couleur = 'black' # Couleur du sommet.
18
19         self.id = ID_SOMMET # À chaque sommet du diagramme de Voronoï est associé un unique identifiant qui servira pour
20             # convertir la géométrie dans un format "adéquat" (cf. travail de mon camarade).
21         ID_SOMMET += 1
22         liste_sommets.append(self)
23
24     def coords(self):
25         '''Renvoyer les coordonnées du sommets.'''
26         return self.x, self.y
27
28     def dessiner(self, rayon = 1, canvas = draw):
29         '''Pour dessiner un sommet sur le canvas fourni.'''
30         dessinerPoints([self.coords()], self.couleur, rayon, canvas)
31
32 class Arete:
33     def __init__(self, arete_soeur = None, arete_suivante = None, arete precedente = None, sommet_but = None, face =
34             None):
35         global ID_ARETES
36         self.arete_soeur = arete_soeur
37         self.arete_suivante = arete_suivante
38         self.arete precedente = arete precedente
39
40         self.sommet_but = sommet_but
41         self.face = face
42
43         self.id = ID_ARETES
44         ID_ARETES += 1
45         liste_aretes.append(self)
46
47     def parcours_dévant(self, f):
48         '''Parcourir les arêtes d'une même face en suivant le sens trigonométrique.'''
49         arete = self
50         while arete.arete_suivante != None and arete.arete_suivante.id != self.id:
51             f(arete)
52             arete = arete.arete_suivante
53         f(arete)
54
55     def dessiner(self, canvas = draw):
56         '''Pour dessiner une arête sur le canvas fourni.'''
57         if self.sommet_but != None and self.arete_soeur.sommet_but != None:
58             canvas.line((self.sommet_but.coords(), self.arete_soeur.sommet_but.coords()), fill = self.couleur)
59
60 class Face:
61     def __init__(self, arete_associee):
62         global ID_FACE
63         self.arete_associee = arete_associee
64
65         self.id = ID_FACE
66         ID_FACE += 1
67         liste_faces.append(self)
68
69     def parcours_profondeur(self, f):
70         '''Parcours en profondeur de la géométrie depuis la face courante.'''
71         faces_visitees = [False] * ID_FACE
72         pile = [self]
73
74         def ajouter_nouvelles_faces(arete):
75             # Permet d'ajouter les faces encore non visitées dans la pile.
76             if arete.arete_soeur.face != None and not faces_visitees[arete.arete_soeur.face.id]:
77                 pile.append(arete.arete_soeur.face)
78
79         while pile != []:
80             face = pile.pop()

```

```

81     f(face) # Retrait d'une face et traitement de cette dernière.
82
83     faces_visitees[face.id] = True # Marquer la face courante comme visitée.
84     face.arete_associee.parcours_devant(ajouter_nouvelles_faces) # Pour ajouter les potentielles nouvelles faces.
85
86 def regrouper(self, face):
87     '''Permet d'incorporer les aretes de la face donnée dans la face courante.'''
88     arete_1, arete_2 = face.arete_associee, face.arete_associee
89     while arete_1.arete_suivante != None and arete_1.arete_suivante.id != face.arete_associee.id:
90         arete_1.face = self
91         arete_1 = arete_1.arete_suivante
92
93     while arete_2.arete_precedente != None and arete_2.arete_suivante.id != face.arete_associee.id:
94         arete_2.face = self
95         arete_2 = arete_2.arete_precedente
96
97 def dessiner(self, canvas = draw):
98     '''Pour dessiner une face sur le canvas fourni.'''
99     self.arete_associee.parcours_devant(lambda arete: arete.dessiner(canvas))
100
101 class Geometrie:
102     def __init__(self, faces):
103         self.faces = faces
104
105     def dessiner(self, canvas = draw):
106         '''Dessiner la géométrie sur le canvas fourni.'''
107         for face in self.faces:
108             face.dessiner(canvas)

```

Puis, ci-contre, le code définissant les breakpoints, les arcs et les deux types d'événements :

```

1 ordonnee = 0 # Ordonnée de la droite de balayage.
2
3 def direction(b):
4     '''Renvoie la direction vers laquelle se dirige le breakpoint b.'''
5     p1, p2 = b.p1, b.p2
6     if p1[0] == p2[0]:
7         if p1[1] < p2[1]:
8             if b.sens == 'g':
9                 return ortho(vecteur(p2, p1))
10            return ortho(vecteur(p1, p2))
11        else:
12            if b.sens == 'g':
13                return ortho(vecteur(p1, p2))
14            return ortho(vecteur(p2, p1))
15    else:
16        if p1[1] > p2[1]:
17            if b.sens == 'g':
18                return ortho(vecteur(p1, p2))
19            return ortho(vecteur(p2, p1))
20        else:
21            if b.sens == 'g':
22                return ortho(vecteur(p2, p1))
23            return ortho(vecteur(p1, p2))
24
25 class BreakPoint:
26     def __init__(self, p1, p2, sens, arete, jumeau = None):
27         '''Intersection des paraboles de foyer p1, p2 et de directrice
28         la droite de balayage. Arête est un pointeur vers l'arête que trace le breakpoint.'''
29         # On s'assure que le point p1 est le site plus à gauche et que p2 est le site le plus à droite.
30         if p1[0] > p2[0]: self.p1, self.p2 = p2, p1 # On place les sites dans l'ordre des x croissants.
31         else: self.p1, self.p2 = p1, p2
32
33         self.sens = sens # Savoir s'il s'agit du BreakPoint gauche ou du BreakPoint droit.
34         # BreakPoint 'jumeau', celui qui trace l'arête soeur du BreakPoint courant.
35         self.jumeau = jumeau
36         self.arete = arete # L'arête tracée par le breakpoint.
37         self.direction = direction(self) # La direction (sous forme de vecteur) que suit le breakpoint.
38
39     def coords(self):
40         '''Renvoie les coordonnées du breakpoint (dépendant de la position de la droite
41         de balayage).'''
42         return intersectionParaboles(self.p1, self.p2, ordonnee, self.sens)
43
44 # Plusieurs arcs attachée à un même site p peuvent apparaître dans l'AVL !
45 class Arc:

```

```

46 def __init__(self, p, voisin_gauche = None, voisin_droit = None, breakpoint_gauche = None, breakpoint_droit = None,
47     → evtcercle = None):
48     '''Représente un arc de parabole défini par le point p(x, y).'''
49     self.p = p
50     self.x, self.y = p
51     self.evtcercle = evtcercle
52     # Pointeur vers les feuilles voisines dans l'AVL.
53     self.voisin_gauche, self.voisin_droit = voisin_gauche, voisin_droit
54     # Pointeur vers les noeuds internes où se trouvent les breakpoints aux extrémités de l'arc.
55     self.breakpoints_gauche, self.breakpoints_droit = breakpoint_gauche, breakpoint_droit
56
57     def coords(self):
58         return self.x, self.y
59
60     # Les événements de la file de priorité, chaque événement possède un indice idx, sa position dans la file de priorité :
61 class EvtPoint:
62     def __init__(self, p):
63         '''p est un point du plan de la forme (x, y).'''
64         self.x, self.y = p
65         self.idx = 0
66
67     def coords(self):
68         return self.x, self.y
69
70 class EvtCercle:
71     def __init__(self, p1, p2, p3, arc):
72         '''p1, p2, p3 trois sites deux-à-deux distincts.'''
73         self.p1, self.p2, self.p3 = p1, p2, p3
74         self.idx = 0
75
76         self.centre, self.r = cercleCirconscrit(p1, p2, p3)
77         # On prend le point du cercle "le plus bas" pour définir l'événement.
78         self.x, self.y = self.centre[0], self.centre[1] + self.r
79         self.arc = arc
80
81     def coords(self):
82         return self.x, self.y

```

On peut maintenant introduire le code de l'arbre binaire équilibré :

```

1 # Implémentation d'un arbre binaire AVL (arbre binaire de recherche équilibré).
2 # Il est primordial (cf. fonction ajouter_arc) que les nuds les plus profond de
3 # l'arbre soient des feuilles : une feuille n'ayant aucun fils.
4
5 def suppression_arc(arc, bg, bd, noeud):
6     '''Cette fonction se charge des formalités lors de la suppression d'un arc.'''
7     nouveau_breakpoint = breakpoint_convergeant(bg, bd, arc.voisin_gauche.p, arc.voisin_droit.p) # Le nouveau breakpoint
8     → à "substituer" dans l'arbre AVL.
9     noeud.etiquette = nouveau_breakpoint
10    # On met à jour les voisins pour les arcs.
11    arc.voisin_gauche.voisin_droit = arc.voisin_droit
12    arc.voisin_droit.voisin_gauche = arc.voisin_gauche
13    arc.voisin_gauche.breakpoints_droit, arc.voisin_droit.breakpoints_gauche = nouveau_breakpoint, nouveau_breakpoint # On
14    → met à jour les breakpoints voisins pour les arcs.
15
16 class AVL:
17     class Noeud:
18         def __init__(self, etiquette, filsG = None, filsD = None):
19             '''Noeud (noeud interne ou feuille).'''
20             self.etiquette = etiquette
21             self.filsG, self.filsD = filsG, filsD
22             self.hauteur = 1
23
24         def estFeuille(self):
25             '''Tester si un nud est une feuille.'''
26             return self.filsG == self.filsD == None
27
28         def filsDroit(self, fils):
29             '''Mettre à jour le fils droit de l'arbre.'''
30             self.filsD = fils
31             self.mettre_hauteur_a_jour()
32
33         def filsGauche(self, fils):
34             '''Mettre à jour le fils gauche de l'arbre.'''
35             self.filsG = fils
36             self.mettre_hauteur_a_jour()

```

```

35
36     def mettre_hauteur_a_jour(self):
37         '''Met à jour la hauteur du noeud courant (normalement, . . .)
38         self.hauteur = 1 + max(self.filsG.hauteur, self.filsD.hauteur)
39
40     def __init__(self, comp = lambda x, y: x < y):
41         '''Arbre binaire AVL.'''
42         self.comp = comp
43
44     def hauteur(self, arb):
45         '''Calcul de la hauteur de l'arbre.'''
46         if arb == None: return 0
47         return arb.hauteur
48
49     def balancement(self, arb):
50         '''Facteur de balancement de l'arbre.'''
51         if arb == None: return 0
52         return self.hauteur(arb.filsG) - self.hauteur(arb.filsD)
53
54     def rotationG(self, arb):
55         '''Rotation à gauche de l'arbre, arb n'est pas une feuille.'''
56         g, d = arb.filsG, arb.filsD
57         if d != None:
58             arb.filsDroit(d.filsG)
59             d.filsGauche(arb)
60             return d
61         return arb
62
63     def rotationD(self, arb):
64         '''Rotation à droite de l'arbre, arb n'est pas une feuille.'''
65         g, d = arb.filsG, arb.filsD
66         if g != None:
67             arb.filsGauche(g.filsD)
68             g.filsDroit(arb)
69             return g
70         return arb
71
72     def reequilibrer(self, arb):
73         '''Ré-équilibrer l'arbre binaire après une insertion/suppression (on fait l'hypothèse que, le balancement vaut ou
74         → bien 2 ou bien -2 : afin de ré-équilibrer un sous-arbre se fasse en temps constant).'''
75         if arb != None:
76             b = self.balancement(arb)
77             if b > 1: # Si le sous-arbre gauche est plus "profond" que le droit.
78                 if self.balancement(arb.filsG) < 0:
79                     arb.filsGauche(self.rotationG(arb.filsG))
80                     return self.rotationD(arb)
81                 if self.balancement(arb.filsD) > 0:
82                     arb.filsDroit(self.rotationD(arb.filsD))
83                     return self.rotationG(arb)
84             return arb
85
86     def rechercher(self, arb, etiquette):
87         '''Recherche de la feuille de l'arbre la "plus proche" de l'étiquette.'''
88         if arb == None or arb.estFeuille():
89             return arb
90
91         if self.comp(etiquette, arb.etiquette):
92             return self.rechercher(arb.filsG, etiquette)
93         else:
94             return self.rechercher(arb.filsD, etiquette)
95
96     def ajouter_arc(self, arb, nouveau_arc):
97         '''Permet d'ajouter un arc de parabole dans l'arbre.'''
98         if arb == None:
99             return self.Noeud(nouveau_arc)
100         elif arb.estFeuille():
101             p1, p2 = arb.etiquette.coords(), nouveau_arc.coords() # Les sites concernés.
102
103             b1, b2 = BreakPoint(p1, p2, 'g', Arete()), BreakPoint(p1, p2, 'd', Arete()) # On construit les breakpoints
104             → (chaque breakpoint "trace" une demi-arête).
105             b1.jumeau, b2.jumeau = b2, b1 # Gestion des breakpoints jumeaux.
106             b1.arete.arete_soeur, b2.arete.arete_soeur = b2.arete, b1.arete # Ces deux breakpoints tracent des demi-arêtes
107             → sur.
108
109             voisin_gauche, voisin_droit = arb.etiquette.voisin_gauche, arb.etiquette.voisin_droit # Gestion des pointeurs
110             → vers les feuilles voisines, on 'recycle' l'arc arc.etiquette.
111             a2, a3 = nouveau_arc, Arc(p1, None, voisin_droit, None, arb.etiquette.breakpoint_droit)
112             if voisin_droit != None:

```

```

110     voisin_droit.voisin_gauche = a3
111     arb.etiquette.voisin_droit, a3.voisin_gauche = a2, a2
112     a2.voisin_gauche, a2.voisin_droit = arb.etiquette, a3
113
114     n1 = self.Noeud(b1, arb, self.Noeud(a2))
115     n2 = self.Noeud(b2, n1, self.Noeud(a3))
116
117     arb.etiquette.breakpoints_droit = b1
118     a3.breakpoints_gauche = b2
119     a2.breakpoints_gauche, a2.breakpoints_droit = b1, b2
120     return n2
121
122 elif self.comp(nouveau_arc, arb.etiquette): # Descente dans l'AVL.
123     arb.filsGauche(self.ajouter_arc(arb.filsG, nouveau_arc))
124 else:
125     arb.filsDroit(self.ajouter_arc(arb.filsD, nouveau_arc))
126
127 return self.reequilibrer(arb) # Ré-équilibrage de l'AVL.
128
129 def supprime_arc(self, arb, etiquette, noeud_g = None, noeud_d = None):
130     '''Permet de supprimer un arc de l'AVL : parmi deux breakpoints qui fusionnent, l'un sera supprimé, l'autre
131     → substitué
132     (il nous faut donc les repérer lors de la descente dans l'AVL).'''
133     if arb.estFeuille():
134         return None # Suppression de l'arc (si l'AVL n'était constitué que d'un seul arc).
135
136     elif self.comp(etiquette, arb.etiquette): # Descente dans l'AVL (à gauche).
137         if arb.filsG.estFeuille(): # On doit supprimer cette feuille ainsi que le nud courant.
138             suppression_arc(arb.filsG.etiquette, noeud_g.etiquette, arb.etiquette, noeud_g)
139             return arb.filsD # Le fils droit est déjà équilibré.
140         else:
141             arb.filsGauche(self.supprime_arc(arb.filsG, etiquette, noeud_g, arb)) # On descend dans le fils gauche en
142             → gardant une trace du dernier BreakPoint 'droit'.
143     else:
144         if arb.filsD.estFeuille(): # On doit supprimer cette feuille ainsi que le nud courant.
145             suppression_arc(arb.filsD.etiquette, arb.etiquette, noeud_d.etiquette, noeud_d)
146             return arb.filsG # Ce sous-arbre est déjà équilibré.
147     else:
148         arb.filsDroit(self.supprime_arc(arb.filsD, etiquette, arb, noeud_d)) # On descend dans le fils droit en gardant
149             → une trace du dernier BreakPoint 'gauche'.
150
151     return self.reequilibrer(arb) # Ré-équilibrage de l'AVL.
152
153 def parcours_prefixe(self, arb, f):
154     '''Parcours préfixe de l'arbre avec traitement de chaque nud.'''
155     if arb != None:
156         f(arb)
157         self.parcours_prefixe(arb.filsG, f)
158         self.parcours_prefixe(arb.filsD, f)

```

Enfin, voici le code qui gère les formalités lors de la convergence de deux breakpoints :

```

1 def direction_convergence(bg, bd, p1, p2):
2     '''Renvoie le sens dans lequel se dirige le nouveau breakpoint issu de la convergence de bg et de bd. Ce nouveau
2     → breakpoint
3     est associé aux sites p1 et p2.'''
4     dir_g, dir_d = normaliser(bg.direction), normaliser(bd.direction) # Direction normalisée des deux breakpoints
4     → convergents.
5     if det([[dir_g[0], dir_d[0]], [dir_g[1], dir_d[1]]]) >= 0: # On s'arrange pour avoir dir1 suivi de dir2 dans le sens
5     → trigonométrique.
6         dir_g, dir_d = dir_d, dir_g
7
8     dir_site = normaliser(ortho(vecteur(p1, p2))) # On prend une direction normalisée, a priori arbitraire, de la
8     → médiatrice des sites p1 et p2.
9     if fdot(dir_g, dir_site) + fdot(dir_site, dir_d) <= pi: # Si cette direction n'est pas dans l'espace formé par les
9     → deux demi-droites (bg, dir1) et (bd, dir2).
10        dir_site = ortho(vecteur(p2, p1))
11
12    if dir_site[0] <= 0: return 'g' # On renvoie la direction du breakpoint.
13    return 'd'
14
15 def breakpoint_convergeant(bg, bd, p1, p2):
16     '''Lorsque les breakpoints bg (breakpoint à gauche du sommet) et bd (breakpoint à droite du sommet) convergent,
17     cette fonction assure la bonne construction du diagramme de Voronoï. Elle renvoie le nouveau breakpoint associé aux
17     → sites
18     p1 et p2 (et construit sont jumeau, qui ne figurera pas dans l'arbre AVL).'''

```

```

19 # Le sommet où il y a convergence :
20 x, y = bg.coords() # Ces coordonnées ne sont pas exactement celle du sommet du diagramme de Voronoï, mais elles sont
21     ↪ très proches.
22 s = Sommet(x, y)
23
24 # Les deux nouvelles demi-arêtes que tracera le nouveau breakpoint issu de la fusion de b1 et de b2.
25 arete_1, arete_2 = Arete(), Arete()
26 arete_1.arete_soeur, arete_2.arete_soeur = arete_2, arete_1
27
28 arete_1.sommet_but = s
29 bg.arete.sommet_but, bd.arete.sommet_but = s, s # On fixe l'origine des arêtes tracées par les deux breakpoints qui
30     ↪ convergent.
31
32 bg.arete.arete_suivante, bd.arete.arete_suivante = bd.jumeau.arete, arete_2 # On fixe l'arête suivante pour le cas
33     ↪ des breakpoints convergeant.
34 bd.jumeau.arete.arete_precedente, arete_2.arete_precedente = bg.arete, bd.arete # On fixe l'arête précédente.
35 bg.jumeau.arete.arete_precedente, arete_1.arete_suivante = arete_1, bg.jumeau.arete # Idem pour les arêtes
36     ↪ manquantes.
37
38 # Construction du nouveau BreakPoint et de son jumeau (il ne sera pas intégré dans l'arbre AVL).
39 sens = direction_convergence(bg, bd, p1, p2)
40 nouveau_b = BreakPoint(p1, p2, sens, arete_2)
41 jumeau_b = BreakPoint(p1, p2, sens, arete_1)
42 nouveau_b.jumeau, jumeau_b.jumeau = jumeau_b, nouveau_b
43
44 # Gestion des faces :
45 if bg.jumeau.arete.face == None:
46     f1 = Face(arete_1)
47     arete_1.face, bg.jumeau.arete.face = f1, f1
48 else:
49     arete_1.face = bg.jumeau.arete.face
50
51 if bd.arete.face == None:
52     f2 = Face(arete_2)
53     arete_2.face, bd.arete.face = f2, f2
54 else:
55     arete_2.face = bd.arete.face
56
57 if bd.jumeau.arete.face == None:
58     if bg.arete.face == None:
59         f3 = Face(bg.arete)
60         bg.arete.face, bd.jumeau.arete.face = f3, f3
61     else:
62         bd.jumeau.arete.face = bg.arete.face
63 else:
64     if bg.arete.face == None:
65         bg.arete.face = bd.jumeau.arete.face
66     elif bg.arete.face.id != bd.jumeau.arete.face.id:
67         bg.arete.face.regrouper(bd.jumeau.arete.face) # Mise en commun des deux faces.
68 return nouveau_b
69
70 ######
71 # Fonctions de manipulation de la géométrie & utilitaire pour le clippage des cellules.
72 def chemin_aretes(liste_points, face_g, face_d, couleur = 'black'):
73     '''Pour une liste de point données, séparant deux faces, renvoie le chemin d'arêtes passant par tous ces points, dans
74     ↪ l'ordre.
75     Attention, les arêtes aux extrémités auront des paramètres incomplets !'''
76     assert len(liste_points) >= 2, 'Pas assez de points !'
77     sommets = [Sommets(p[0], p[1]) for p in liste_points]
78
79     a_g = [Arete(None, None, None, sommets[1], face_g)]
80     a_d = [Arete(None, None, None, sommets[0], face_d)]
81     a_g[0].arete_soeur, a_d[0].arete_soeur = a_d[0], a_g[0]
82
83     for i in range(1, len(liste_points) - 1):
84         arete_g = Arete(None, None, a_g[-1], sommets[i + 1], face_g)
85         arete_d = Arete(None, a_d[-1], None, sommets[i], face_d)
86         arete_g.arete_soeur, arete_d.arete_soeur = arete_d, arete_g
87
88         a_g[-1].arete_suivante, a_d[-1].arete_precedente = arete_g, arete_d
89
90         a_g.append(arete_g)
91         a_d.append(arete_d)
92
93     return a_g, a_d
94
95 def clipper_demi_droite(p, v, c = (0, 0), L = longueur, l = largeur):
96     '''Renvoie le point d'intersection entre la demi-droite d'origine p et de vecteur directeur v avec le rectangle
97     de coin supérieur gauche c, de longueur L (sens des x croissants: de la gauche vers la droite) et de largeur l

```

```

93     (sens des y croissant : du haut vers le bas).''''
94     assert v != [0, 0] and estDansRectangle(p), 'Données invalides pour le clippage !'
95     x, y = c
96     # Cas de demi-droites verticales ou horizontales.
97     if v[0] == 0:
98         y1 = y if v[1] > 0 else y + 1
99         return (p[0], y1)
100    elif v[1] == 0:
101        x1 = x if v[0] > 0 else x + L
102        return (x1, p[1])
103    else: # Cas de demi-droites 'obliques'.
104        if v[1] < 0: # Coin supérieur gauche.
105            p_inter = intersectionDroites(p, v, c, (1, 0))
106        else: # Coin inférieur gauche.
107            p_inter = intersectionDroites(p, v, (x, y + 1), (1, 0))
108
109        if 0 <= p_inter[0] <= L:
110            return p_inter
111        elif v[0] > 0:
112            return intersectionDroites(p, v, (x + L, y), (0, 1))
113        return intersectionDroites(p, v, c, (0, 1))

```

Et finalement, voici les instructions principales pour la construction du diagramme de Voronoï :

```

1 def verifier_convergence(arc_g, arc_d, tas):
2     '''Gestion de la convergence/divergence de breakpoints (on donne deux arcs : un arc gauche pour l'étude de la
3     → convergence à gauche,
4     idem pour l'arc droit (mais à droite)).'''
5     # Triplet gauche, s'il y a :
6     g1 = arc_g.voisin_gauche
7     if g1 != None:
8         g2 = g1.voisin_gauche
9         if g2 != None:
10            ajouter_evenement_circulaire(g2, g1, arc_g, tas)
11
12     # Triplet droit, s'il y a :
13     d1 = arc_d.voisin_droit
14     if d1 != None:
15         d2 = d1.voisin_droit
16         if d2 != None:
17             ajouter_evenement_circulaire(arc_d, d1, d2, tas)
18
19 def ajouter_evenement_circulaire(arc_g, arc_m, arc_d, tas):
20     '''Ajouter, si nécessaire, un événement circulaire dans le tas.'''
21     p1, p2, p3 = arc_g.coords(), arc_m.coords(), arc_d.coords() # Les sites (normalement distincts).
22     if p1 != p3: # Dans le cas contraire, les deux breakpoints associés à ce triplet d'arcs divergent.
23         b1, b2 = arc_m.breakpoint_gauche, arc_m.breakpoint_droit
24         if intersecteDemiDroite(b1.coords(), b1.direction, b2.coords(), b2.direction):
25             evt = EvtCercle(p1, p2, p3, arc_m)
26             arc_m.evtcercler = evt # On marque l'arc avec ce nouvel événement circulaire.
27             tas.ajouter(evt) # On ajoute cet événement au tas.
28
29 def retirer_evenement_circulaire(arc, tas):
30     '''Retire l'événement circulaire associé à l'arc donné du tas.'''
31     if arc.evtcercler != None:
32         tas.extraire(arc.evtcercler.idx) # Retrait de l'événement circulaire (en général, c'est une 'fausse alerte').
33         arc.evtcercler = None # On retire l'événement circulaire.
34
35     # Gestion des événements :
36     def evenement_ponctuel(tas, arb, racine, evt):
37         '''Prise en charge d'un événement ponctuel.'''
38         nouvel_arc = Arc(evt.coords())
39         if racine == None:
40             racine = arb.ajouter_arc(racine, nouvel_arc)
41         else:
42             arc = arb.rechercher(racine, evt).etiquette
43             retirer_evenement_circulaire(arc, tas) # Suppression de la "fausse alerte".
44
45             racine = arb.ajouter_arc(racine, nouvel_arc) # Ajout de l'arc de parabole.
46             verifier_convergence(nouvel_arc, nouvel_arc, tas) # Vérifier la convergence des nouveaux breakpoints (ajout
47             → d'événements circulaires si besoin).
48
49     def evenement_circulaire(tas, arb, racine, evt):
50         '''Prise en charge d'un événement circulaire.'''
51         arc = evt.arc

```

```

51 voisin_gauche, voisin_droit = arc.voisin_gauche, arc.voisin_droit
52 # On supprime les potentiels événements circulaires qui auraient pu se produire avec les arcs voisins de l'arc
53 # concerné.
54 retirer_evenement_circulaire(voisin_gauche, tas)
55 retirer_evenement_circulaire(voisin_droit, tas)

56 # Coordonnées du point au milieu du segment joignant les deux breakpoints (pour retrouver l'arc à supprimer).
57 x, y = (arc.breakpoint_gauche.coords()[0] + arc.breakpoint_droit.coords()[0]) / 2, evt.coords()[1]
58 racine = arb.supprime_arcs(racine, Arc((x, y))) # Suppression de l'arc.
59 verifier_convergence(voisin_droit, voisin_gauche, tas) # Vérifier la convergence des breakpoints (ajout d'événements
# circulaires si besoin).
60 return racine

61 # La fonction principale :
62 def voronoi(P):
63     '''On suppose les points deux-à-deux distincts.'''
64     global ordonnee
65     n = len(P)
66     # Cas particuliers (non traités) : n = 0 et n = 1.
67
68     compTas = lambda p1, p2 : p1.coords()[:-1] < p2.coords()[:-1] # Comparaison lexicographique suivant la coordonnée y
# (balayage de haut en bas) puis x (si besoin).
69     compArb = lambda p1, p2 : p1.coords() < p2.coords() # Comparaison lexicographique suivant la coordonnée x (ligne de
# front, de gauche à droite) puis y (si besoin).
70
71     tas = TasMin(3*n - 5, [EvtPoint(p) for p in P], compTas, True)
72     arb, racine = AVL(compArb), None
73
74     # On balaye le plan du haut vers le bas, étant donné le repérage, on descend dans le sens des y croissants.
75     while not tas.estVide():
76         evt = tas.minimum()
77         ordonnee = evt.y # On met à jour l'ordonnée de la droite de balayage.
78         if isinstance(evt, EvtPoint):
79             racine = evenement_punctuel(tas, arb, racine, evt) # Événement ponctuel.
80         else:
81             ordonnee -= 1e-20 # Pour éviter les risques d'erreurs liés aux imprécisions numériques (valeur expérimentale).
82             racine = evenement_circulaire(tas, arb, racine, evt) # Événement circulaire.

83
84     # À l'issu, les breakpoints qui restent dans l'arbre AVL correspondent à des demi-droites, on procède en deux temps :
85     # -> d'abord on va clipper ces demi-droites avec les bords de la fenêtre et finaliser la construction des arêtes
# correspondantes (via un parcours en profondeur de l'arbre AVL),
86     # -> puis, on parcourt la géométrie (grâce à un parcours en profondeur à l'aide des faces et des arêtes) afin de :
87     #   i) finaliser la construction des faces (en fermant les cellules non bornées avec les bords de la fenêtre),
88     #   ii) recenser toutes les faces.
89     def traitement_demi_droites(noeud):
90         b = noeud.etiquette
91         if isinstance(b, BreakPoint):
92             c, d = b.coords(), b.jumeau.arete.sommet_but.coords()
93             if not estDansRectangle(c) and estDansRectangle(d):
94                 p = clipper_demi_droite(d, b.direction)
95                 s = Sommet(p[0], p[1])
96                 b.arete.sommet_but = s
97
98         arb.parcours_prefixe(racine, traitement_demi_droites)
99         faces = []
100        if ID_FACE == 0: # Ceci atteste que le diagramme de Voronoï est dégénéré et ne contient que des droites (i.e. tous les
# sites sont alignés).
101            pass
102        else: # Le diagramme de Voronoï est non dégénéré (il contient seulement des segments et des demi-droites comme
# les arêtes).
103            def traitement_faces(face):
104                # clipper_face(face) # Je n'ai pas eu le temps de couper proprement les cellules avec les bords de la fenêtre.
105                faces.append(face)
106                # Comme n >= 2, le diagramme de Voronoï, dans ce cas, contient au moins une demi-droite donc, la racine de l'arbre
# → AVL est un BreakPoint.
107                racine.etiquette.arete.face.parcours_profondeur(traitement_faces)
108            return Geometrie(faces)
109
110    def decoupage():
111        grp = regrouper(sante, alimentaire)
112        evn = list(map(enveloppe_convexe, grp)) # Calcul de la liste des enveloppes convexes (deux-à-deux disjointes).
113        sites = list(map(isobarycentre, evn)) # Les sites.
114
115        # dessiner_enveloppe_convexe(evn) # Dessin des enveloppes convexes.
116        dessinerPoints(alimentaire, 'red', 2)
117        dessinerPoints(sante, 'green', 2)
118        dessinerPoints(sites, 'blue', 3)
119
120        # Construction du diagramme de Voronoï avec les isobarycentres des enveloppes convexes.
121

```

```
122     return voronoi(sites)
```

## Ajuster le tracé des zones aux routes : codes

Ci-dessous se trouve les codes pour l'extraction et la construction du graphe des routes :

```

1 import json
2 import xml.etree.ElementTree as ET
3
4 ##### Calcul de la distance entre deux points à la surface de la Terre.
5 def distance_reelle(p1, p2, r = RT):
6     '''Renvoie la 'distance réelle' entre deux points (latitude, longitude) à une altitude r (typiquement, le rayon de la
7     → Terre).
8     On utilise la formule de haversine.'''
9     # Conversion en radian des latitudes et des longitudes des deux points.
10    lat1, lon1, lat2, lon2 = radians(p1[0]), radians(p1[1]), radians(p2[0]), radians(p2[1])
11    # Formule de haversine (on procède en deux temps).
12    expr = sin((lat2 - lat1) / 2) ** 2 + cos(lat1) * cos(lat2) * sin((lon2 - lon1) / 2) ** 2
13    # On arrondi à 5 décimales après la virgule (gain de mémoire).
14    return round(2 * r * asin(sqrt(expr)), 5)
15
16 #####
17
18 # Extraction des composantes du graphe des routes.
19 def extraire():
20     '''Extraire les noeuds et routes du fichier de donnée XML.'''
21     fichier_xml = "../Data/CentreVilleMap.xml"
22     arbre = ET.iterparse(fichier_xml)
23     noeuds, noeuds_utiles, routes = {}, {}, [] # On ne garde que les noeuds "utiles".
24
25     # Parcours du fichier XML.
26     # Le fichier XML est structuré grossièrement en deux blocs :
27     # -> le premier contenant tous les noeuds,
28     # -> le second contient toutes les routes.
29     for (evt, elm) in arbre:
30         if elm.tag == 'node': # Gestion d'un noeud.
31             attributs = elm.attrib
32             noeuds[attributs['id']] = (float(attributs['lat']), float(attributs['lon']))
33         elif elm.tag == 'way': # Gestion d'une route.
34             if any([n.attrib['k'] == 'highway' and n.attrib['v'] not in ['path', 'cycleway', 'footway', 'steps', 'elevator']
35             → for n in elm.findall('tag'))]:
36                 lst = []
37                 for n in elm.findall('nd'):
38                     ref = n.attrib['ref']
39                     lst.append(ref)
40                     noeuds_utiles[ref] = noeuds[ref]
41                 routes.append(lst)
42             # On renvoie les noeuds et les routes.
43             return noeuds_utiles, routes
44 #####
45
46 # Pour charger les composantes géographiques du centre-ville (noeuds et routes) ainsi que le graphe.
47 def charger_composantes():
48     with open('routes.json', 'r') as fichier: routes = json.loads(fichier.read())
49     with open('noeuds.json', 'r') as fichier: noeuds = json.loads(fichier.read())
50     return routes, noeuds
51
52 def charger_graphe():
53     with open('graphe.json', 'r') as fichier: graphe = json.loads(fichier.read())
54     return graphe
55
56 def construireGraphe():
57     '''Construit le graphe pondéré à partir des fichiers "nuds.txt" et "routes.txt".'''
58     # On adopte une représentation sous forme de dictionnaire d'adjacence :
59     # -> on conserve les identifiants des noeuds enregistrés.
60     # -> les poids correspondent à la distance réelle entre les points.
61     graphe = dict()
62     routes, noeuds = charger_composantes()
63
64     # Construction du graphe :
65     for r in routes:
66         for i in range(len(r) - 1):
```

```

67     n1, n2 = r[i], r[i + 1]
68     d = distance_reelle(noeuds[n1], noeuds[n2]) # Distance entre les deux points : le poids de l'arête.
69
70     if graphe.get(n1) == None: graphe[n1] = {n2 : d}
71     else: graphe[n1][n2] = d
72
73     if graphe.get(n2) == None: graphe[n2] = {n1 : d}
74     else: graphe[n2][n1] = d
75
76     # On renvoie le graphe.
77     return graphe
78
79 def dessinerGraphe(couleur_sommet = 'blue', couleur_arete = 'black', canvas = draw):
80     '''Dessine le graphe des routes sur la carte.'''
81     routes, noeuds = charger_composantes()
82
82     for r in routes:
83         for i in range(len(r) - 1):
84             n1, n2 = r[i], r[i + 1]
85             p1, p2 = polaire2pixel(noeuds[n1]), polaire2pixel(noeuds[n2])
86
87             canvas.line((p1, p2), fill = couleur_arete)
88             dessinerPoints([p2], couleur_sommet, 1, canvas)
89
90     dessinerPoints([polaire2pixel(noeuds[r[0]])], couleur_sommet, 1, canvas)

```

Et voici les codes pour l'extraction des composantes connexes ainsi que l'algorithme de Dijkstra pour la recherche d'un plus court chemin :

```

1 ######
2 # Fonctions pour la manipulation de graphes.
3 def composantesConnexes(graphe):
4     '''Renvoie la liste des composantes connexes du graphe donnée (sous forme de dictionnaire).
5     La complexité de cette fonction est : temporelle O(a + s), spatiale O(a + s) pour s sommets et a arêtes.'''
6     composantes_connexes, deja_vus = [], dict()
7
8     def parcourt_profondeur(sommet):
9         '''Parcours en profondeur du graphe depuis un sommet donné.'''
10        composante, pile = dict(), [sommet]
11        while len(pile) > 0:
12            s = pile.pop() # On prend le sommet au dessus de la pile.
13            if not deja_vus.get(s, False): # S'il n'a pas encore été visité.
14                deja_vus[s] = True # On marque s comme sommet visité.
15                composante[s] = graphe[s] # On met à jour la composante connexe en construction.
16                pile.extend(graphe[s].keys()) # On met tous les voisins de s dans la pile.
17
18        # On renvoie la composante connexe, elle est vide (i.e. égale à dict()) lorsque sommet a déjà été visité.
19        return composante
20
21    for sommet in graphe.keys(): # Pour chaque sommet du graphe.
22        composante = parcourt_profondeur(sommet) # On extrait la composante connexe dont il fait parti.
23        if composante != dict(): # Si le sommet n'a pas déjà été visité, on ajoute la composante connexe dans la liste.
24            composantes_connexes.append(composante)
25
26    return composantes_connexes # On renvoie la liste des composantes connexes.
27 #####
28
29 # Algorithme de Dijkstra.
30 def dijkstra(debut, graphe, fin = None):
31     '''Mise en uvre de l'algorithme de Dijkstra pour la recherche d'un plus court chemin
32     du sommet 'debut' au sommet 'fin' dans le graphe fourni.
33     Le graphe est un dictionnaire : {id noeud : {id voisin : distance, ...}}.
34     Renvoie le dictionnaire des distances et des prédecesseurs ; une condition d'arrêt est
35     possible si le sommet 'fin' est précisé.
36     Sa complexité est : temporelle O((a + s) * log(s)), spatiale O(s) avec s sommets et a arêtes.'''
37     # Cette classe permet "d'accélérer" l'algorithme de Dijkstra en facilitant la mise à jour du tas binaire.
38     class Sommet:
39         def __init__(self, sommet):
40             self.sommet = sommet # Identifiant du sommet.
41             self.idx = 0 # Position dans le tas-min.
42
43         # Dictionnaire de correspondance (identifiant du sommet : objet Sommet).
44         distance, predecesseur, correspondance = dict(), dict(), dict()
45         sommets = graphe.keys() # Les sommets du graphe.
46         n = len(sommets) # Nombre de sommets du graphe.
47         # On autorise l'indexation sur les objets du tas pour faciliter leur manipulation.
48         tas = TasMin(n, [], lambda s1, s2 : distance[s1.sommet] < distance[s2.sommet], True)
49
50         for s in sommets: # Initialisation des structures de données.

```

```

49     distance[s], correspondance[s] = graphe[debut].get(s, float('inf')), Sommet(s)
50     if s != debut:
51         tas.ajouter(correspondance[s]) # La file de priorité.
52
53     for e in range(n - 2):
54         S = tas.minimum() # On retire du tas le sommet S dont on est sûr de connaître un plus court chemin de 'debut' à S.
55         if S.sommet == fin: # Pour terminer dès que le sommet 'fin' a été rencontré.
56             return distance, predecesseur
57
58     for v in graphe[S.sommet].keys(): # On restreint la recherche aux voisins de S seulement.
59         d = distance[S.sommet] + graphe[S.sommet][v]
60         if d < distance[v]: # On regarde si la nouvelle distance du sommet 'debut' au sommet v est plus courte.
61             distance[v], predecesseur[v] = d, S.sommet # On met à jour, pour le sommet v, ses données.
62             tas.percole_haut(correspondance[v].idx) # Préserver l'invariant grâce à sa position dans le tas, mémorisée.
63     # On renvoie le dictionnaire des distances et des prédecesseurs.
64     return distance, predecesseur
65
66 def construireChemin(debut, fin, predecesseur, noeuds):
67     '''Étant donné deux sommets debut et fin ainsi qu'un "dictionnaire des prédecesseurs",
68     renvoie la liste ordonnée des identifiants des sommets pour aller de "debut" à "fin".
69     Attention : on fait ici l'hypothèse qu'un chemin entre "debut" et "fin" existe !'''
70     liste, sommet = [], fin
71     while sommet != None: # Tant qu'il existe des prédecesseurs.
72         liste.append(polaire2pixel(noeuds[sommet])) # On ajoute le prédecesseur dans la liste.
73         sommet = predecesseur.get(sommet, None) # On regarde s'il possède lui-aussi un prédecesseur ('None' sinon).
74     # On renvoie le chemin ordonné, du début à la fin.
75     return [polaire2pixel(noeuds[debut])] + liste[::-1]

```

Finalement, voici les codes pour ajuster le diagramme de Voronoï aux routes :

```

1 # Dictionnaire de correspondance entre les identifiants des sommets du diagramme de Voronoï et
2 # les identifiants des sommets du graphe (après ajustement des premiers).
3 sommets_graphe = dict()
4
5 # Ajuster les arêtes du diagramme de Voronoï aux routes.
6 def ajusterSommet(sommets, graphe, noeuds):
7     '''Cette fonction altère les coordonnées des sommets du diagramme de Voronoï en
8     remplaçant chaque sommet par le sommet du graphe routier le plus proche.'''
9     p = {polaire2pixel(noeuds[nom]): nom for nom in graphe.keys()}
10    arb = Arbre2D(list(p.keys())) # Un peu lourd...
11    racine = arb.construire(0, len(p), 0)
12
13    for sommet in sommets:
14        voisin = arb.plusProcheVoisin(racine, sommet.coords(), 0)
15        sommet.x, sommet.y = voisin[0], voisin[1]
16        sommets_graphe[sommet.id] = p[voisin]
17
18 def ajusterAretes(aretes, graphe, noeuds):
19     '''Cette fonction altère les arêtes du diagramme de Voronoï afin de les
20     faire coïncider avec les routes.
21     Attention : on fait ici l'hypothèse que les sommets du diagramme de Voronoï
22     coïncident avec des sommets du graphe.'''
23     arete_ajustees = [False] * ID_ARETES # Mémoriser les arêtes déjà traitées.
24     for i in range(ID_ARETES):
25         arete = aretes[i]
26         if not arete_ajustees[arete.id]:
27             if arete.sommet_but != None and arete.arete_soeur.sommet_but != None:
28                 # Marquer l'arête et sa soeur comme ajustées.
29                 arete_ajustees[arete.id], arete_ajustees[arete.arete_soeur.id] = True, True
30
31             n1, n2 = sommets_graphe[arete.sommet_but.id], sommets_graphe[arete.arete_soeur.sommet_but.id]
32             distance, predecesseur = dijkstra(n1, graphe, n2)
33             if distance[n2] <= 1.5 * distance_reelle(noeuds[n1], noeuds[n2]):
34                 chemin = construireChemin(n1, n2, predecesseur, noeuds)
35                 chemin_g, chemin_d = chemin_aretes(chemin, arete.arete_soeur.face, arete.face)
36
37                 # On recoule les nouveaux chemins aux arêtes déjà présentes.
38                 chemin_d[-1].arete_precedente, arete.arete_precedente.arete_suivante = arete.arete_precedente, chemin_d[-1]
39                 chemin_d[0].arete_suivante, arete.arete_suivante.arete_precedente = arete.arete_suivante, chemin_d[0]
40
41                 chemin_g[0].arete_precedente, arete.arete_soeur.arete_precedente.arete_suivante =
42                     arete.arete_soeur.arete_precedente, chemin_g[0]
43                 chemin_g[-1].arete_suivante, arete.arete_soeur.arete_suivante.arete_precedente =
44                     arete.arete_soeur.arete_suivante, chemin_g[-1]
45             else:
46                 arete.couleur, arete.arete_soeur.couleur = 'red', 'red'

```

```

46 def ajusterVoronoi(graphe, noeuds):
47     '''Ajuster le diagramme de Voronoï au graphe routier.'''
48     ajusterSommet(liste_sommets, graphe, noeuds) # Ajuster les sommets du diagramme de Voronoï.
49     ajusterAretes(voronoi, graphe, noeuds) # Ajuster les arêtes du diagramme de Voronoï.
50 #####
51
52 # Extraction et mémorisation des routes et des noeuds :
53 noeuds, routes = extraire() # charger_composantes()
54 with open('noeuds.json', 'w+') as fichier: fichier.write(json.dumps(noeuds))
55 with open('routes.json', 'w+') as fichier: fichier.write(json.dumps(routes))
56
57 # Construction, extraction de la plus grosse composante connexe et mémorisation du graphe des routes :
58 graphe = construireGraphe(noeuds, routes) # charger_graphe()
59 liste_composantes = composantesConnexes(graphe)
60 tri_fusion(liste_composantes, 0, None, lambda x, y : len(x.keys()) < len(y.keys()))
61 graphe = liste_composantes[-1]
62 with open('graphe.json', 'w+') as fichier: fichier.write(json.dumps(graphe))
63
64 # Construction du diagramme de Voronoï et ajustement aux routes :
65 voronoi = decoupage()
66 ajusterVoronoi(graphe, noeuds)
67 voronoi.dessiner()
68 carte.save('decoupage.png')

```

## Références

- [1] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [2] Mark Berg, Otfried Cheong, Marc Kreveld, and Mark Overmars. *Computational Geometry*. Springer Berlin, Heidelberg, 2008.
- [3] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209226, sep 1977.
- [4] Dinesh P. Mehta and Sartaj Sahni. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, New York, 2017.