Rapport

May 3, 2022

1 Jouer avec la géométrie, le Jeu de Voronoï

1.1 Projet Python 1A ENSAE

1.1.1 Paul Wattellier et Kacim Younsi

[1]:
```
from IPython.display import Image
```

Introduction :

Notre projet implémente le jeu de Voronoï qui s'appuie sur le concept des diagrammes de Voronoï. Un diagramme de Voronoï est un pavage du plan en cellules à partir d'un ensemble discret de points appelés « germes ». Chaque cellule enferme un seul germe, et forme l'ensemble des points du plan plus proches de ce germe que d'aucun autre. Une arête d'un tel graphe est alors appelée arête de Voronoï et un sommet est appelé nœud de Voronoï.

Dans le jeu de Voronoï chaque joueur place successivement un point sur le plateau de jeu et obtient un score déterminé par la surface des cellules qu'il a placé, l'objectif est d'obtenir le plus de surface à la fin du jeu. Dans notre projet nous jouerons avec 5 points pour chacun des joueurs, nous jouerons à un jeu de Voronoï solo dans lequel l'adversaire est un ordinateur.

[13]:
```
Image("https://i.ibb.co/HB7hcYc/voronoileg.png")
```

[13]:

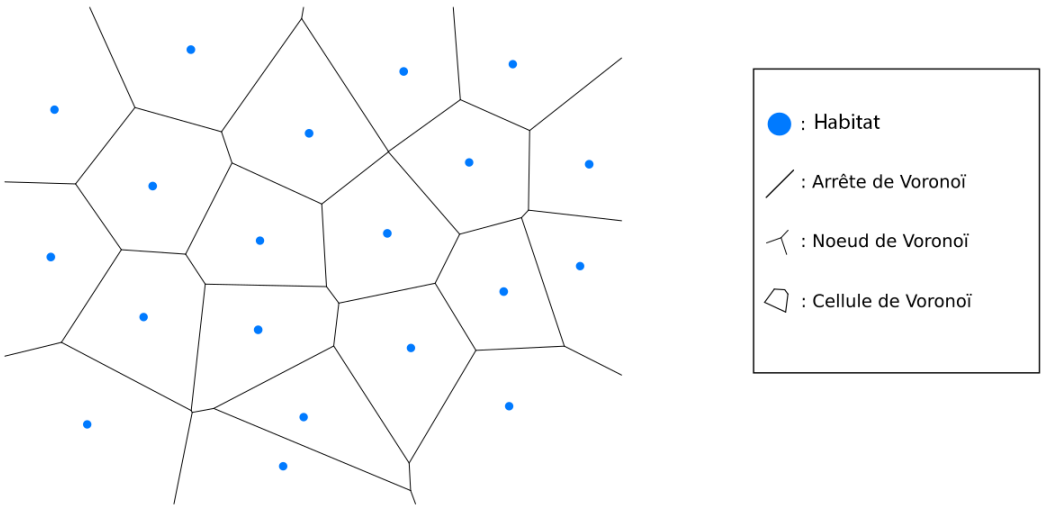

Figure 1 - Exemple de diagramme de Voronoï (source: fait main sur inkscape comme les autres graphes 2D)

2 1. Tracé du diagramme de Voronoï

2.1 1.1. Algorithme de Fortune

Divers algorithmes permettent de réaliser un diagramme de Voronoï, il existe un algorithme incrémental simple mais coûteux en temps, un algorithme de type diviser pour régner, compliqué à mettre en place mais rapide et un algorithme plus astucieux qui possède la même complexité que l'algorithme du type diviser pour régner mais qui est plus simple à mettre en place, cet algorithme du nom de son créateur est celui que nous étudierons donc ici.

2.1.1 1.1.1. L'idée originale et élégante de Fortune

L'algorithme de Fortune est un algorithme de Balayage, une droite de balayage balaie la carte et trace le diagramme au cours du balayage. Cependant une approche directe ne fonctionnera pas ici car comme on peut le voir sur l'illustration une telle droite fait face à la présence d'évènements imprévisibles, le diagramme dépend de points situés au-dessous de la droite.

[6]:
```
Image("https://i.ibb.co/dbLyQm0/evti3.png")
```
[6]:

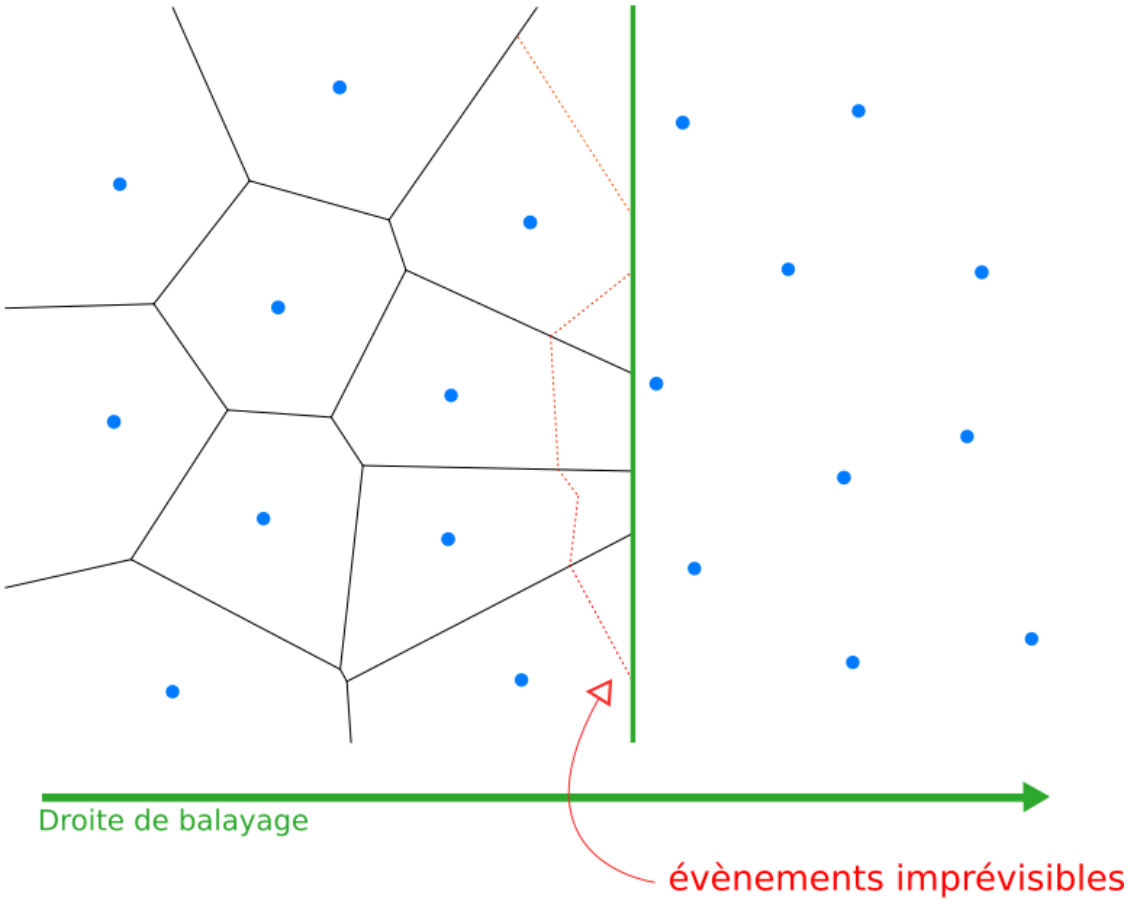

Figure 2 - Algorithme de balayage classique

L'idée superbe et élégante de Fortune pour remédier à ce problème et d'ajouter une 3ème dimension.

[13]:
```
Image("https://i.ibb.co/zH3r362/conejeu.png")
```
[13]:

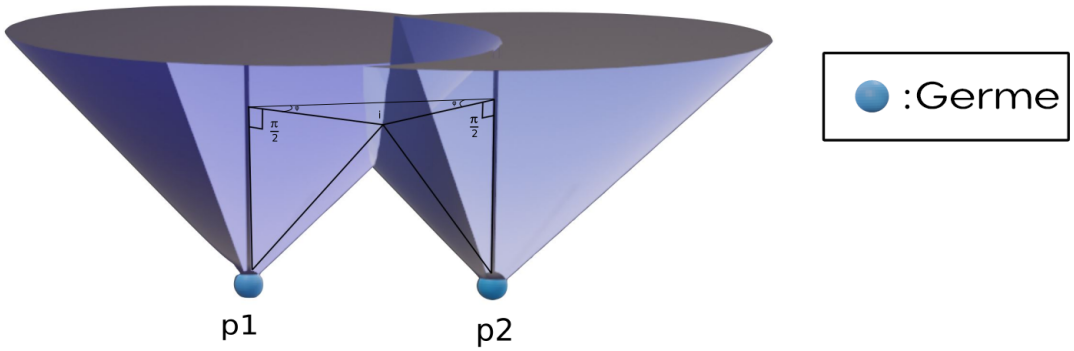

Figure 3 - cônes (source: fait main sur blender comme les autres figures 3D)

On pose c1 et c2 deux cônes d'angle $\frac{\pi}{4}$ respectivement associés aux germes p1 et p2. Soit i un point d'intersection de c1 et c2. On observe que i est situé sur la face extérieur de chaque cône à une hauteur h et i est donc à la distance $hsin(\frac{\pi}{4}) = \frac{h}{\sqrt{2}}$ de chaque sommet et il est donc équidistant de chaque sommet et le projeté sur le plan horizontal est de même à équidistance des 2 germes.

[14]:
```
Image("https://i.ibb.co/zh23LHd/3dinters.png")
```

[14]:

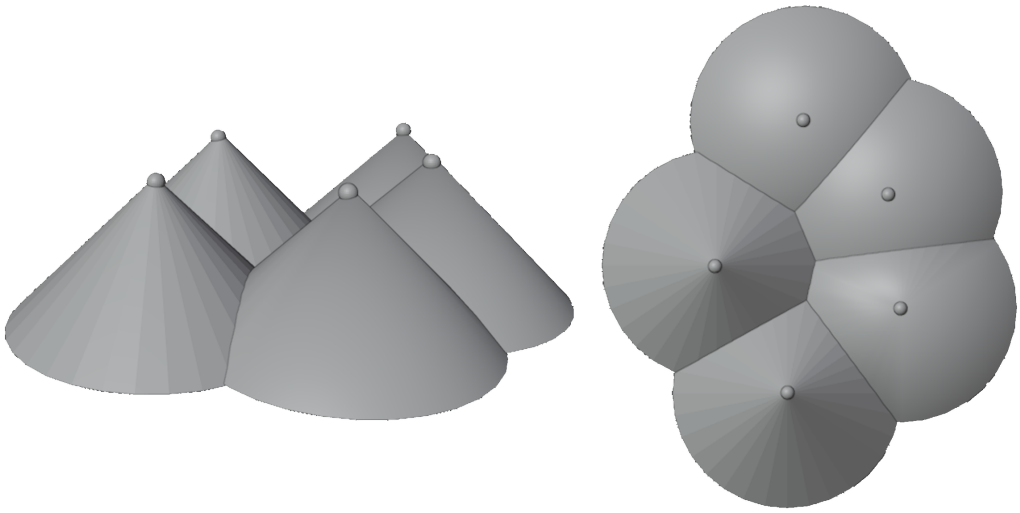

Figure 4 - Principe de l'algorithme de Fortune

Ainsi en intersectant tous les cônes associés à chaque germe du plan, les cellules de Voronoï se forment au niveau de l'intersection d'un cône avec ses cônes voisins d'après la propriété d'équidistance et le projeté sur le plan horizontal permet l'obtention du diagramme de Voronoï.

[15]:
```
Image("https://i.ibb.co/5Gg3wvm/3Ddeux.png")
```

[15]:

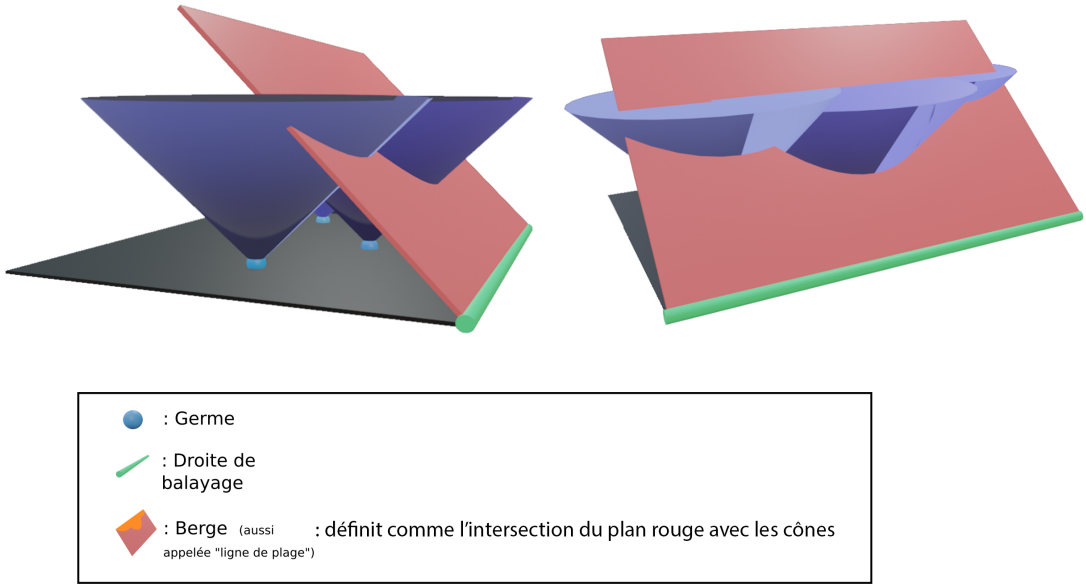

Figure 5 - Algorithme de Fortune

Notons →x et →y les directions du plan, avec →x la direction de balayage. A chaque germe pi, on associe le cône Ci de sommet pi, d'axe de troisième direction →z et d'angle /4. Fortune a l'idée d'associer à la droite de balayage un plan incliné d'un angle /4 identique à celui des cônes dans la direction opposée à celle de balayage. Il nomme alors le projeté sur la carte de l'intersection de ce plan avec les cônes : la berge, celle-ci représente un front parabolique qui balaie la carte d'où le nom de berge. Tous les points de la berge sont à équidistance de la droite de balayage et du germe le plus proche.

[16]:
```
Image("https://i.ibb.co/rbwtKNN/bergedist.png")
```

[16]:

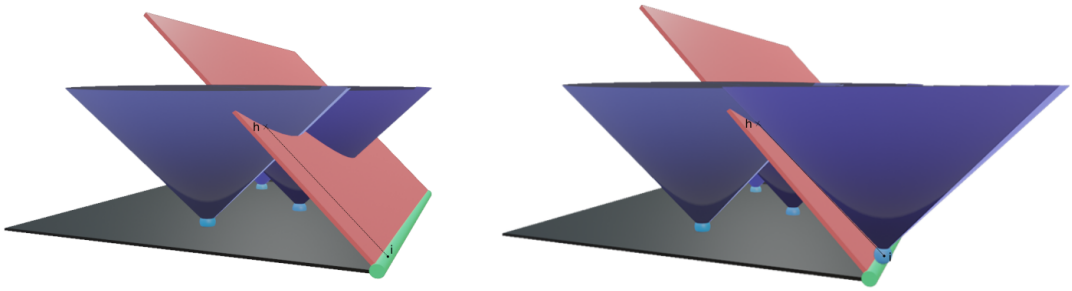

Figure 6 - Principe d'équidistance de la berge à la droite de balayage et la germe la plus proche

En effet considérons par exemple ce point h de la berge dont la germe la plus proche est noté p. Le chemin le plus cours reliant h à la berge est naturellement un segment dans la direction x de

balayage on le note $[h, i]$, en considérant un cône ci associé au point i d'angle pi/4 on observe que ce cône intersecte le cône associé à p en h. Donc h est à équidistance de p et de la berge d'après le résultat précédent sur l'intersection de 2 cônes de même angle.

[7]:
```
Image("https://i.ibb.co/QMbBNf1/3deuxrupture.png")
```

[7]:

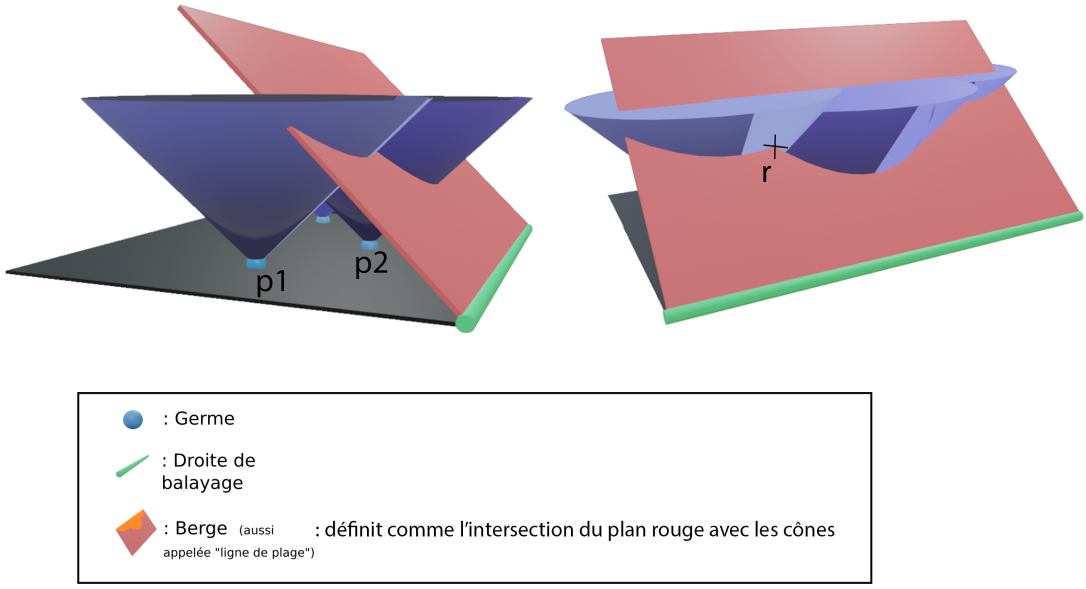

Figure 7 - Point de rupture du front parabolique

Ainsi les points de rupture des paraboles de la berge sont situés sur les arrêtes du diagramme de Voronoï car ils se trouvent à égale distance de 2 germes adjacentes et les germes qui n'ont pas étés vu par la droite de balayage sont situés à une plus grande distance de la berge que tout autre point déjà vu, ainsi la cellule de Voronoï hypothétique appartenant à ces points n'atteindra la berge que lorsque ces points seront vu par la droite de balayage. Le front parabolique permet donc la définition complète du diagramme de Voronoï car il permet d'obtenir toutes les arêtes du diagramme. On peut à présent repasser sur une vision en 2D pour comprendre l'évolution du front parabolique.

2.1.2 1.1.2. Comprendre l'évolution du front parabolique

On suppose que la coordonné selon x d'un germe est une clé primaire. Le nombre de paraboles que comporte la berge peut diminuer ou augmenter cela se produit respectivement lors d'évènement dit ponctuels et circulaires.

[18]:
```
Image("https://i.ibb.co/xL0R9cQ/ponctuelevt.png")
```

[18]:

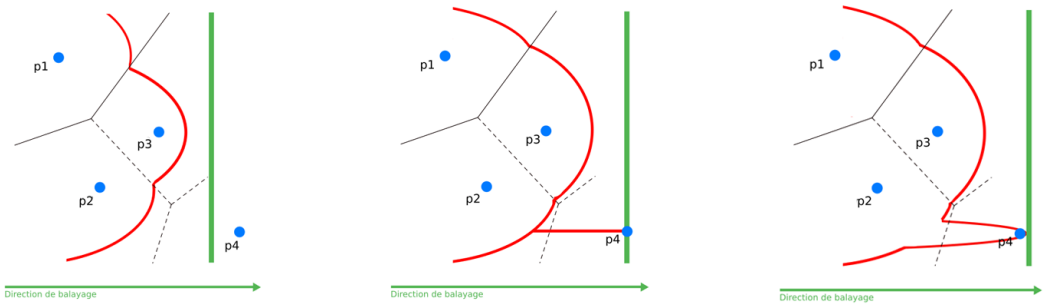

Figure 8 - Evènement ponctuel

Lorsqu'un point, p4 ici, apparait au niveau de la ligne de balayage on parle d'évènement ponctuel, une droite s'ajoute à la berge comme on le voie en projetant le segment $[i, h]$ précédent sur la carte. Mais dès lors que la droite de balayage est à une distance strictement supérieure à p4.x on observe qu'une parabole complète est ajoutée à la berge on observe donc qu'une parabole de l'ancien front est scindé en deux et qu'une nouvelle parabole est ajoutée au front, on a donc au total l'ajout de 2 nouvelles paraboles au front.

[19]:
```
Image("https://i.ibb.co/LNPpQ90/circulaireevt.png")
```

[19]:

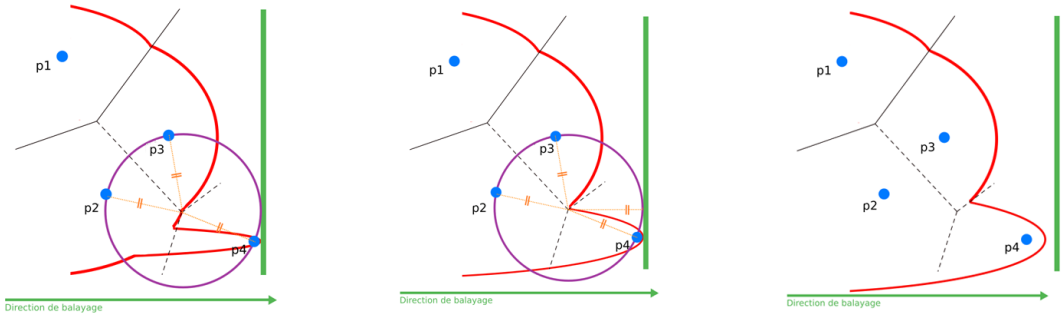

Figure 9 - Evènement circulaire

Le seul moyen pour une parabole de disparaître est d'être derrière 2 paraboles, lorsque cela se produit on a donc un point intersectant 3 paraboles, par définition de ces paraboles ce point se trouve donc à équidistance de 3 germes adjacentes et il s'agit donc d'un nœud du diagramme de Voronoï. Lorsqu'un point apparait on vérifie s'il y a apparition d'un évènement circulaire, ce qui signifie, on teste s'il existe un cercle circonscrit à 3 germes dont les paraboles appartiennent à la berge et sont consécutives, parmi lesquelles se trouve le germe rencontré, on vérifie aussi qu'aucun autre germe du diagramme ne se trouve à l'intérieur d'un tel cercle sinon nous n'avons aucune information. Dans ce cas particulier il y a alors disparition d'une parabole.

Ainsi un ensemble discret d'évènement permettent de tracer le diagramme de Voronoï en simulant une approche continue d'un algorithme de balayage.

2.1.3 1.1.3. Complexité d'un tel algorithme

On s'intéresse à présent à la complexité d'un tel algorithme. En représentant le front parabolique par un arbre binaire de recherche équilibré les ajouts et les suppressions se font alors en $\mathcal{O}(\log n)$ si le nombre de sommets du diagramme est un $\mathcal{O}(n)$ on a donc un $\mathcal{O}(n)$ évènements circulaires et comme on a n évènements ponctuels on déduit que le temps de calcul dans le pire cas est en $\mathcal{O}(n \log n)$ car il y a $\mathcal{O}(n)$ évènements qui viennent modifier le front parabolique chacun est traité en $\mathcal{O}(\log n)$ ne dépendant pas de la taille du diagramme et le front permet d'établir le diagramme de Voronoï. Voyons donc que l'on a $\mathcal{O}(n)$ sommets sur un diagramme de Voronoï. Nous allons pour cela utiliser la relation d'Euler généralisée, mais celle-ci n'étant valable que pour des surfaces compactes, il faut d'abord transformer le diagramme de Voronoï.

[20]:
```
Image("https://i.ibb.co/fkbr6wy/homeo.jpg")
```

[20]:

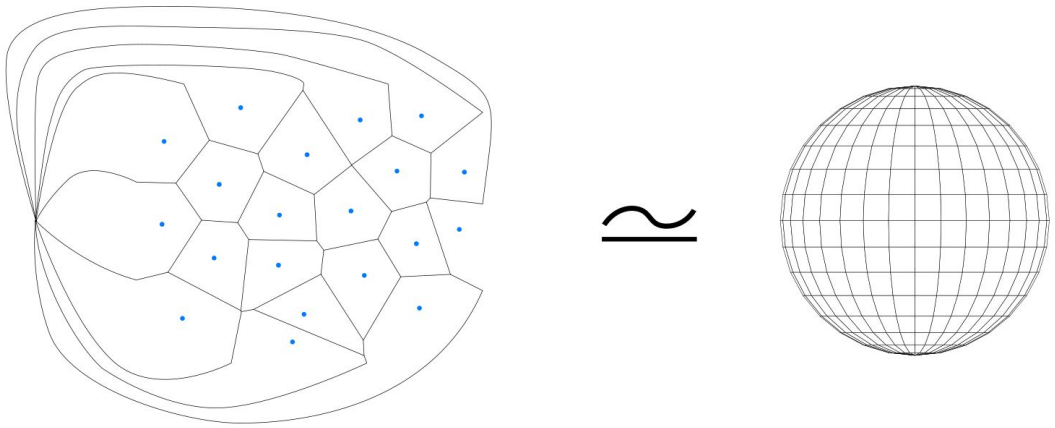

Figure 10 - Homéomorphisme entre un diagramme et une sphère

Ajoutons virtuellement un sommet de Voronoï à l'infini et relions-y toutes les arêtes de Voronoï non bornés (voir figure x) : on a transformé le plan en surface homéomorphe à une sphère, donc la relation d'Euler nous dit que $\chi_{Vor} = 2$. De plus, par définition $\chi_{Vor} = V^{'} - E + F$, avec F le nombre de cellules de Voronoï du diagramme étendu et $V^{'}$ son nombre de sommets :$V^{'} = V + 1$ et $F = n$, puisqu'on a ajouté un sommet sans créer ni supprimer de cellule. Comme chaque arête de Voronoï a exactement deux sommets aux extrémités, on a $2E = \sum_i degresommet(s_i)$ (le degré d'un sommet est son nombre de voisins). Si les germes ne sont pas alignés, on sait par définition que chaque sommet correspond à l'intersection de trois cellules, donc est voisin d'au moins 3 autres sommets, ce qui se traduit par $\sum_i degre(s_i) \geq 3(V + 1)$. On a donc 2E 3(V + 1) et V + 1 − E + n = 2, d'où V 2n − 5 et E 3n − 6. Donc le nombre de sommet V est bien un $\mathcal{O}(n)$. La complexité de l'algorithme est donc en $\mathcal{O}(n \log n)$.

2.1.4 1.2 Implémentation d'un tel algorithme sur Python

Pour l'implémentation de l'algorithme de Fortune sur Python nous nous somme aidés de l'algorithme de Janson Hendryli (https://github.com/jansonh/Voronoi) qui est l'implémentation python du code C++ de Matt Brubeck. Nous avons ensuite largement édité ce code afin de corriger quelques bugs et d'implémenter le jeu de Voronoi au complet. Dans cette partie nous allons

voir comment est implémenté l'algorithme de Fortune au sein de la version finale de notre code.

On commence par définir l'objet Voronoï, la boîte dans laquelle on va pouvoir placer nos germes puis tracer le diagramme.

[ ]:
```
class Voronoi:
    def __init__(self, points, player=Player(1), bot=Player(0)):
        self.player = Player(1)
        self.bot = Player(0)
        self.output = []  # liste des segments qui forment le diagramme de␣
↪Voronoï
        self.arc = None  # arbre binaire pour les paraboles
        self.points = PriorityQueue()  # événements ponctuels
        self.points_save = []
        self.event = PriorityQueue()  # événements circulaires

        # On pose la boîte dans laquelle on trace la diagramme (bounding box)
        self.x0 = 0.0
        self.x1 = 0.0
        self.y0 = 0.0
        self.y1 = 0.0

        # Insertion des points en tant qu'évènements ponctuels
        for pts in points:
            if pts[2] == 0:
                point = Point(pts[0], pts[1], player=self.bot)
            else:
                point = Point(pts[0], pts[1], player=self.player)
            self.points_save.append(point)
            self.points.push(point)
            # On agrandit la boîte si nécessaire
            if point.x < self.x0:
                self.x0 = point.x
            if point.y < self.y0:
                self.y0 = point.y
            if point.x > self.x1:
                self.x1 = point.x
            if point.y > self.y1:
                self.y1 = point.y
        # on ajoute des marges de sécurité
        dx = (self.x1 - self.x0 + 1) / 5.0
        dy = (self.y1 - self.y0 + 1) / 5.0
        self.x0 = self.x0 - dx
        self.x1 = self.x1 + dx
        self.y0 = self.y0 - dy
        self.y1 = self.y1 + dy
```

On définit ensuite la méthode permettant de tracer le diagramme

[ ]:
```
    def process(self):
        while not self.points.empty():
            if not self.event.empty() and (self.event.top().x <= self.points.
↪top().x):
                self.process_event()  # gère les évènements circulaires
            else:
                self.process_point()  # gère les évènements ponctuels

        # après avoir traité les points les évènements circulaires restant sont␣
↪traités
        while not self.event.empty():
            self.process_event()

        self.finish_edges()
```

Cependant cette méthode fait appel à plein d'autres méthodes qui restent indéfinies. On commence alors par définir ce que signifie traiter les évènements ponctuels et circulaires :

[ ]:
```
    def process_point(self):
        # on récupère l'évènement ponctuel de la file de priorité
        p = self.points.pop()
        # ajoute la nouvelle parabole au front parabolique
        self.arc_insert(p)

    def process_event(self):
        # on récupère l'évènement circulaire de la file de priorité
        e = self.event.pop()

        if e.valid:
            # début d'un nouveau segment
            s = Segment(e.p, p1=e.p0, p2=e.p1)
            self.output.append(s)

            # on enlève la parabole "engloutie"
            a = e.a
            if a.pprev is not None:
                a.pprev.pnext = a.pnext
                a.pprev.s1 = s
            if a.pnext is not None:
                a.pnext.pprev = a.pprev
                a.pnext.s0 = s

            # termine les segments qui atteignent le noeud
            if a.s0 is not None:
                a.s0.finish(e.p)
            if a.s1 is not None:
                a.s1.finish(e.p)
```

```
            # on regarde alors si cette intervention a permit la création␣
↪d'autres évènements circulaires
            if a.pprev is not None:
                self.check_circle_event(a.pprev, e.x)
            if a.pnext is not None:
                self.check_circle_event(a.pnext, e.x)
```

A nouveau pour avancer dans l'algorithme et définir des méthodes simples nous avons à nouveau fait appel à d'autres méthodes non définies. Nous avons donc définit la méthode arc_insert(self, p)

[ ]:
```
    def arc_insert(self, p):  # beachline
        #print("Ajout d'une parabole pour p: x =", p.x,",y =", p.y)
        if self.arc is None:
            self.arc = Arc(p)
        else:
            # trouve l'arc parabolique à actualiser
            i = self.arc
            while i is not None:
                flag, z = self.intersect(p, i)
                if flag:
                    # nouvelle parabole qui intersecte le front i
                    flag, zz = self.intersect(p, i.pnext)
                    if (i.pnext is not None) and (not flag):
                        i.pnext.pprev = Arc(
                            i.p, player=i.p.player, a=i, b=i.pnext)
                        i.pnext = i.pnext.pprev
                    else:
                        i.pnext = Arc(i.p, player=i.p.player, a=i)
                    i.pnext.s1 = i.s1

                    # ajout de p entre les paraboles i.p et i.pnext
                    i.pnext.pprev = Arc(p, player=p.player, a=i, b=i.pnext)
                    i.pnext = i.pnext.pprev

                    i = i.pnext  # on actualise alors i

                    # on ajoute alors les segments qui sont formés comme vu␣
↪dans la partie théorique
                    seg = Segment(z, p1=p, p2=i.p)
                    self.output.append(seg)
                    i.pprev.s1 = i.s0 = seg

                    seg = Segment(z, p1=p, p2=i.pprev.p)
                    self.output.append(seg)
                    i.pnext.s0 = i.s1 = seg
```

```
                    # on n'oublie pas de regarder si nous avons engendrer des␣
↪évènements circulaires
                    self.check_circle_event(i, p.x)
                    self.check_circle_event(i.pprev, p.x)
                    self.check_circle_event(i.pnext, p.x)

                    return

                i = i.pnext

            # si p n'intersecte pas le front on l'ajoute à la liste
            i = self.arc
            while i.pnext is not None:
                i = i.pnext
            i.pnext = Arc(p, player=p.player, a=i)

            # on ajoute un nouveau segment entre p et et i
            x = self.x0
            y = (i.pnext.p.y + i.p.y) / 2.0
            start = Point(x, y)

            seg = Segment(start, p1=p, p2=i.p)
            i.s1 = i.pnext.s0 = seg
            self.output.append(seg)
```

On a ici besoin de la methode intersect et des méthodes ont été laissées de côtés précédemment, les choses se compliquent. On note de plus que nous faisons appel aux méthodes de certains objets extérieurs tels que Arc (notre front parabolique) et Segment. Ces objets ont été construits à l'occasion et constamment améliorés au cours de notre progression. Pour simplifier les futures lectures nous vous donnons ci-dessous les différents objets utilisés dans leur version finale.

[ ]:
```
class Player:
    polygons = []
    n = None

    def __init__(self, n, score=0):
        self.n = n
        self.polygons = []
        self.score = 0

    def add_pol(self, pol):
        self.polygons.append(pol)


class Point:
    x = 0.0
    y = 0.0
```

```
    def __init__(self, x, y, player=None):
        self.x = x
        self.y = y
        self.player = player

    def distance(self, p):
        return math.sqrt((self.x-p.x)**2 + (self.y-p.y)**2)


class Event:
    x = 0.0
    p = None
    a = None
    valid = True

    def __init__(self, x, p, a, p0, p1):
        self.x = x
        self.p = p
        self.a = a
        self.p0 = p0
        self.p1 = p1
        self.valid = True


class Arc:
    p = None
    pprev = None
    pnext = None
    e = None
    s0 = None
    s1 = None

    def __init__(self, p, player=None, a=None, b=None):
        self.player = player
        self.p = p
        self.pprev = a
        self.pnext = b
        self.e = None
        self.s0 = None
        self.s1 = None
# Arc(p) définit la parabole associé au point p


class Segment:
    start = None
    end = None
```

```
    done = False
    p1 = None
    p2 = None
    score1 = False
    score2 = False

    def __init__(self, p, p1=None, p2=None):
        self.start = p
        self.end = None
        self.done = False
        self.p1 = p1
        self.p2 = p2

    def finish(self, p, edge=False):
        if not edge:
            if self.done:
                return
            self.end = p
            self.done = True
        else:
            self.end = p
            self.done = True

    def point(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def hauteur(self, p):
        if self.end.x - self.start.x == 0:
            p_inter = Point(self.start.x, p.y)
        if self.end.y - self.start.y == 0:
            p_inter = Point(p.x, self.start.y)
        elif self.end.x - self.start.x != 0:
            a1 = (self.end.y - self.start.y) / (self.end.x - self.start.x)
            # pente d'une droite perpendiculaire à self
            a2 = -1/a1
            # abscisse du point d'intersection des deux droites
            x0 = (-a1*(self.start.x) + a2*(p.x) - (p.y) + (
                self.start.y))/(a2 - a1)
            y0 = a2*(x0 - p.x) + p.y
            p_inter = Point(x0, y0)

        return p.distance(p_inter)

    def actu_score(self):
        if self.p1 is not None and not self.score1:
            self.p1.player.score += self.hauteur(self.p1)*(
```

```
                self.start.distance(self.end))/2
            self.score1 = True
        if self.p2 is not None and not self.score1:
            self.p2.player.score += self.hauteur(self.p2)*(
                self.start.distance(self.end))/2
            self.score2 = True

    def p_edge(self, p):
        # renvoie l'intersection du Segment self avec le bord qui est dépassé␣
↪par le segment
        if self.start.x - self.end.x != 0:
            a = (self.end.y - self.start.y)/(self.end.x - self.start.x)
            x2 = 500
            y2 = a*(500 - self.start.x) + self.start.y
            if 0 < y2 < 500 and p.x > self.p1.x:
                return Point(x2, y2)

            x2 = 0
            y2 = a*(0 - self.start.x) + self.start.y
            if 0 < y2 < 500 and p.x < self.p1.x:
                return Point(x2, y2)

            y2 = 500
            x2 = (500)/a - self.start.y/a + self.start.x
            if 0 < x2 < 500 and p.y > self.p1.y:
                return Point(x2, y2)

            y2 = 0
            x2 = (-self.start.y)/a + self.start.x
            if 0 < x2 < 500 and p.y < self.p1.y:
                return Point(x2, y2)
        elif p.y > 500:
            return Point(p.x, 500)
        else:
            return Point(p.x, 0)

    def inter_edge(self):
        # regarde si un Segment dépasse un bord, si oui remet son extrémité sur␣
↪le bord du canvas
        if self.start.x < 0 or self.start.x > 500 or self.start.y < 0 or self.
↪start.y > 500:
            self.start = self.p_edge(self.start)

        if self.end.x < 0 or self.end.x > 500 or self.end.y < 0 or self.end.y >␣
↪500:
            self.end = self.p_edge(self.end)
```

arc_insert tout comme process_event fait tout de même appel à une méthode non définie : check_circle_event. Cette méthode est assez astucieuse car nous regardons si un évènement circulaire à lieu pour un front parabolique donné à un abscisse de balayage x0. Cette fonction nous suffit car en vertu de la théorie nous savons exactement en quels x0 nos évènements circulaires ont lieu.

[ ]:
```
    def check_circle_event(self, i, x0):
        # regarde si un nouvel évènement circulaire va se produire sur le front
↪i lorsque la droite de balayage va atteindre x0
        if (i.e is not None) and (i.e.x != self.x0):
            i.e.valid = False
        i.e = None

        if (i.pprev is None) or (i.pnext is None):
            return

        flag, x, o = self.circle(i.pprev.p, i.p, i.pnext.p)
        if flag and (x > self.x0):
            i.e = Event(x, o, i, i.pprev.p, i.pnext.p)
            self.event.push(i.e)
```

Ici nous n'avons pas encore tout à fait fini car l'on a besoin de faire un peu de trigonométrie avec la méthode circle et de revenir sur intersect :

[ ]:
```
    def circle(self, a, b, c):
        if ((b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)) > 0:
            return False, None, None

        # Joseph O'Rourke, Computational Geometry in C (2nd ed.) p.189
        A = b.x - a.x
        B = b.y - a.y
        C = c.x - a.x
        D = c.y - a.y
        E = A*(a.x + b.x) + B*(a.y + b.y)
        F = C*(a.x + c.x) + D*(a.y + c.y)
        G = 2*(A*(c.y - b.y) - B*(c.x - b.x))

        if (G == 0):
            return False, None, None  # Points alignés

        # point o au centre du cercle
        ox = 1.0 * (D*E - B*F) / G
        oy = 1.0 * (A*F - C*E) / G

        # o.x plus le rayon égale max x coord
        x = ox + math.sqrt((a.x-ox)**2 + (a.y-oy)**2)
        o = Point(ox, oy)
```

```
        return True, x, o
```

[ ]:
```
    def intersect(self, p, i):
        # regarde si la parabole associé à p intersect l'arc i
        if (i is None):
            return False, None
        if (i.p.x == p.x):
            return False, None

        a = 0.0
        b = 0.0

        if i.pprev != None:
            a = (self.intersection(i.pprev.p, i.p, 1.0*p.x)).y
        if i.pnext != None:
            b = (self.intersection(i.p, i.pnext.p, 1.0*p.x)).y

        if (((i.pprev is None) or (a <= p.y)) and ((i.pnext is None) or (p.y <=
↪b))):
            py = p.y
            px = 1.0 * ((i.p.x)**2 + (i.p.y-py)**2 -
                        p.x**2) / (2*i.p.x - 2*p.x)
            res = Point(px, py)
            return True, res
        return False, None

    def intersection(self, p0, p1, l):
        # donne l'intersection des paraboles associés à p0(l) et p1(l) où l est
↪l'abscisse de la droite de balayage
        p = p0
        if (p0.x == p1.x):
            py = (p0.y + p1.y) / 2.0
        elif (p1.x == l):
            py = p1.y
        elif (p0.x == l):
            py = p0.y
            p = p1
        else:
            z0 = 2.0 * (p0.x - l)
            z1 = 2.0 * (p1.x - l)

            a = 1.0/z0 - 1.0/z1
            b = -2.0 * (p0.y/z0 - p1.y/z1)
            c = 1.0 * (p0.y**2 + p0.x**2 - l**2) / z0 - \
                1.0 * (p1.y**2 + p1.x**2 - l**2) / z1

            py = 1.0 * (-b-math.sqrt(b*b - 4*a*c)) / (2*a)
```

```
        px = 1.0 * (p.x**2 + (p.y-py)**2 - l**2) / (2*p.x-2*l)
        res = Point(px, py)
        return res
```

Dernière méthode pour finir : finish_edges qui termine les segments qui intersectent les bords.

[ ]:
```
    def finish_edges(self):
        l = self.x1 + (self.x1 - self.x0) + (self.y1 - self.y0)
        i = self.arc
        while i.pnext != None:
            if i.s1 != None:
                p = self.intersection(i.p, i.pnext.p, l*2.0)
                i.s1.finish(p)
            i = i.pnext

        #nous verrons plus tard cette seconde partie elle est affichée ici car␣
↪nous montrons les versions finales de chaques partie du code pour ne pas␣
↪perdre le lecteur
        self.clean_output()
        for s in self.output :
            s.inter_edge()


        Ls_edge = self.correct_belonging()
```

3 2. Extension de l'algorithme pour l'implémentation du jeu de Voronoï

La deuxième grande étape de notre projet a été d'améliorer cet algorithme afin d'implémenter le jeu de Voronoï. Pour cela nous avons commencé par créer l'objet Player et l'attribut player de l'objet Point (vus précédemment) afin que l'algorithme puisse distinguer les germes du joueur 1 et du joueur 2. Une fois la classe implémentée une partie du travail reste à faire au niveau de l'interface graphique (savoir quel joueur joue quand on double clique sur le canvas et distinguer visuellement les cellules des joueurs en coloriant les surfaces qu'elles occupent) mais le gros du travail va être de calculer le score de chaque joueur.

Pour calculer le score d'un joueur on cherche à mesurer l'aire de chacune de ses cellules. Pour identifier clairement les cellules nous avons eu l'idée d'ajouter les attributs p1 et p2 à chaque Segment afin d'associer à chaque segment les germes des cellules adjacentes. Ainsi en parcourant l'ensemble des segments et regardant tout ceux associés à une germe p nous pouvons former la liste des segments qui composent la cellule p. En parcourant alors la liste des germes chacune étant associé à un joueur il est facile d'obtenir la liste des cellules de chaque joueur.

Une fois la liste des cellules obtenue pour mesurer l'aire des cellule nous avons 2 méthodes. Ou bien nous regardons l'ensemble des aires assoicées aux segments à l'aide de la formule $\frac{(base)(hauteur)}{2}$ ou bien nous construisons les polygones associés aux cellules et calculons l'aire des polygones à l'aide d'un module.

Nous avons commencé par implémenté la première méthode c'est donc celle dont nous parlerons en premier

3.1 2.1. La méthode des segments

Un premier problème avec cette méthode est le fait que nous n'avons pas accès aux segments au bord du Canvas. L'étape la plus importante est ici de construire une fonction qui récupère ces segments. Mais commencons par les tâches faciles.

Tout d'abord la méthode finish_edges ne termine pas les segments aux bord du canvas mais plus loin. Nous avons donc codé une petite méthode permettant de corriger ce défaut facilement.

[ ]:
```
def p_edge(self, p):
        # renvoie l'intersection du Segment self avec le bord qui est dépassé␣
↪par le segment
        if self.start.x - self.end.x != 0:
            a = (self.end.y - self.start.y)/(self.end.x - self.start.x)
            x2 = 500
            y2 = a*(500 - self.start.x) + self.start.y
            if 0 < y2 < 500 and p.x > self.p1.x:
                return Point(x2, y2)

            x2 = 0
            y2 = a*(0 - self.start.x) + self.start.y
            if 0 < y2 < 500 and p.x < self.p1.x:
                return Point(x2, y2)

            y2 = 500
            x2 = (500)/a - self.start.y/a + self.start.x
            if 0 < x2 < 500 and p.y > self.p1.y:
                return Point(x2, y2)

            y2 = 0
            x2 = (-self.start.y)/a + self.start.x
            if 0 < x2 < 500 and p.y < self.p1.y:
                return Point(x2, y2)
        elif p.y > 500:
            return Point(p.x, 500)
        else:
            return Point(p.x, 0)

    def inter_edge(self):
        # regarde si un Segment dépasse un bord, si oui remet son extrémité sur␣
↪le bord du canvas
        if self.start.x < 0 or self.start.x > 500 or self.start.y < 0 or self.
↪start.y > 500:
            self.start = self.p_edge(self.start)
```

```
        if self.end.x < 0 or self.end.x > 500 or self.end.y < 0 or self.end.y >␣
↪500:
            self.end = self.p_edge(self.end)
```

Une fois cela fait nous avons créé des fonctions permettant d'accéder à la hauteur de chaque triangle formé par les extrémités d'un segment et le point p.

[23]:
```
    def hauteur(self, p):
        if self.end.x - self.start.x == 0:
            p_inter = Point(self.start.x, p.y)
        if self.end.y - self.start.y == 0:
            p_inter = Point(p.x, self.start.y)
        elif self.end.x - self.start.x != 0:
            a1 = (self.end.y - self.start.y) / (self.end.x - self.start.x)
            # pente d'une droite perpendiculaire à self
            a2 = -1/a1
            # abscisse du point d'intersection des deux droites
            x0 = (-a1*(self.start.x) + a2*(p.x) - (p.y) + (
                self.start.y))/(a2 - a1)
            y0 = a2*(x0 - p.x) + p.y
            p_inter = Point(x0, y0)

        return p.distance(p_inter)
```

Alors nous sommes en mesure de créer la méthode qui va attribuer les scores de chaque joueur lorsqu'on la fait boucler sur l'ensemble des segments.

[24]:
```
    def actu_score(self):
        if self.p1 is not None and not self.score1:
            self.p1.player.score += self.hauteur(self.p1)*(
                self.start.distance(self.end))/2
            self.score1 = True
        if self.p2 is not None and not self.score1:
            self.p2.player.score += self.hauteur(self.p2)*(
                self.start.distance(self.end))/2
            self.score2 = True
```

Pour que celle-ci soit fonctionnelle il faut tout de même que la liste des segments soit bien établie et il nous reste donc à accéder aux segments au bord du Canvas. On réalise donc pour ce faire deux méthodes, la première next_edge(self,n,direction,s,s_edge) qui retourne à un segment s qui intersecte le bord en s_edge le segment s_new qui représente le segment qui part de s_edge qui avance dans la direction : direction et qui n'est pas intersecté par aucun autre segment. De plus cette fonction renvoie corner qui est un tuple booléen,Point qui indique si le segment à atteint un corner et si oui celui-ci est retourné. Pour que cette méthode fonctionne on donne l'information supplémentaire du bord sur lequel on avance $n \in \{1, 2, 3, 4\}$.

[26]:
```
```

```
    def next_edge(self, n, direction, s, s_edge):  # fonction qui a un segment␣
↪s ayant pour extremité s_edge sur le bord n du canvas trouve le segment␣
↪ininterrompu qui longe c bord dans le direction : direction (=1 pour montée␣
↪=0 pour descente en regardant le bord gauche)
        s_new = Segment(s_edge)
        corner = (False, Point(-1, -1))
        if n == 1:
            assert(s_edge.x == 500)  # bord gauche du canvas
            Ls = []
            Ly = []
            for s2 in self.output:
                if direction == 1:
                    b2 = s2.start.y > s_edge.y
                else:
                    b2 = s2.start.y < s_edge.y

                if s2.start.x == 500 and b2:  # si un segment interromp le␣
↪longement du bord on le note
                    Ls.append((s2, 's'))
                    Ly.append(s2.start.y)

                if direction == 1 and s2.end != None:
                    b2 = s2.end.y > s_edge.y
                elif s2.end != None:
                    b2 = s2.end.y < s_edge.y

                if s2.end != None and s2.end.x == 500 and b2:
                    Ls.append((s2, 'e'))
                    Ly.append(s2.end.y)

            if len(Ls) > 0:
                if direction == 1:
                    s0, str = Ls[Ly.index(min(Ly))]
                else:
                    s0, str = Ls[Ly.index(max(Ly))]
                if str == 's':
                    s_new.finish(s0.start)
                else:
                    s_new.finish(s0.end)
            else:
                if direction == 1:
                    s_new.finish(Point(500, 500))
                    corner = (True, Point(500, 500))
                else:
                    s_new.finish(Point(500, 0))
                    corner = (True, Point(500, 0))
```

```
            # s_new est donc le segment qui va de s_edge au premier point qui␣
↪interromp le parcours du bord depuis s_edge dans la direction : direction

        if n == 2:
            assert(s_edge.y == 500)
            Ls = []
            Lx = []
            for s2 in self.output:
                if direction == 1:
                    b2 = s2.start.x < s_edge.x
                else:
                    b2 = s2.start.x > s_edge.x

                if s2.start.y == 500 and b2:  # si un segment interromp le␣
↪longement du bord on le note
                    Ls.append((s2, 's'))
                    Lx.append(s2.start.x)

                if direction == 1 and s2.end != None:
                    b2 = s2.end.x < s_edge.x
                elif s2.end != None:
                    b2 = s2.end.x > s_edge.x

                if s2.end != None and s2.end.y == 500 and b2:
                    Ls.append((s2, 'e'))
                    Lx.append(s2.end.x)

            if len(Ls) > 0:
                if direction == 1:
                    s0, str = Ls[Lx.index(max(Lx))]
                else:
                    s0, str = Ls[Lx.index(min(Lx))]
                if str == 's':
                    s_new.finish(s0.start)
                else:
                    s_new.finish(s0.end)
            else:
                if direction == 1:
                    s_new.finish(Point(0, 500))
                    corner = (True, Point(0, 500))
                else:
                    s_new.finish(Point(500, 500))
                    corner = (True, Point(500, 500))

        if n == 3:
            assert(s_edge.x == 0)
            Ls = []
```

```
            Ly = []
            for s2 in self.output:
                if direction == 1:
                    b2 = s2.start.y < s_edge.y
                else:
                    b2 = s2.start.y > s_edge.y

                if s2.start.x == 0 and b2:  # si un segment interromp le␣
↪longement du bord on le note
                    Ls.append((s2, 's'))
                    Ly.append(s2.start.y)

                if direction == 1 and s2.end != None:
                    b2 = s2.end.y < s_edge.y
                elif s2.end != None:
                    b2 = s2.end.y > s_edge.y

                if s2.end != None and s2.end.x == 0 and b2:
                    Ls.append((s2, 'e'))
                    Ly.append(s2.end.y)

            if len(Ls) > 0:
                if direction == 1:
                    s0, str = Ls[Ly.index(max(Ly))]
                else:
                    s0, str = Ls[Ly.index(min(Ly))]
                if str == 's':
                    s_new.finish(s0.start)
                else:
                    s_new.finish(s0.end)
            else:
                if direction == 1:
                    s_new.finish(Point(0, 0))
                    corner = (True, Point(0, 0))
                else:
                    s_new.finish(Point(0, 500))
                    corner = (True, Point(0, 500))

        if n == 4:
            assert(s_edge.y == 0)
            Ls = []
            Lx = []
            for s2 in self.output:
                if direction == 1:
                    b2 = s2.start.x > s_edge.x
                else:
                    b2 = s2.start.x < s_edge.x
```

```
                if s2.start.y == 0 and b2:  # si un segment interromp le␣
↪longement du bord on le note
                    Ls.append((s2, 's'))
                    Lx.append(s2.start.x)

                if direction == 1 and s2.end != None:
                    b2 = s2.end.x < s_edge.x
                else:
                    b2 = s2.end.x > s_edge.x

                if s2.end != None and s2.end.y == 0 and b2:
                    Ls.append((s2, 'e'))
                    Lx.append(s2.end.x)

            if len(Ls) > 0:
                if direction == 1:
                    s0, str = Ls[Lx.index(min(Lx))]
                else:
                    s0, str = Ls[Lx.index(max(Lx))]
                if str == 's':
                    s_new.finish(s0.start)
                else:
                    s_new.finish(s0.end)
            else:
                if direction == 1:
                    s_new.finish(Point(500, 0))
                    corner = (True, Point(500, 0))
                else:
                    s_new.finish(Point(0, 0))
                    corner = (True, Point(0, 0))

        p_center = Point(s_new.start.x/2 + s_new.end.x/2,
                         s_new.start.y/2 + s_new.end.y/2)
        d1 = s.p1.distance(p_center)
        d2 = -1
        if s.p2 is not None:
            d2 = s.p2.distance(p_center)
        if d2 > 0 and d2 < d1:
            s_new.p1 = s.p2
        else:
            s_new.p1 = s.p1

        return s_new, corner
```

Une fois cette fonction définit on peut à présent faire notre grande boucle qui va nous permettre d'accéder à la liste des segments aux bords. On rappelle tout de même qu'on aurait ici pu faire plus de fonctions intermédiaires afin d'avoir un code plus compact

[ ]:
```
    def correct_seg(self):

        Ls_edge = []

        for s in self.output:
            if s.start.x == 500:
                s_new, b_corner = self.next_edge(1, 0, s, s.start)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k = 0
                n0 = 1
                dir = -1
                while bool and k < 4:
                    n0 += dir
                    n0 = n0 % 4
                    if n0 == 0:
                        n0 = 4
                    s_new, b_corner = self.next_edge(n0, 0, s, corner)
                    Ls_edge.append(s_new)
                    bool, corner = b_corner
                    k += 1

                s_new, b_corner = self.next_edge(1, 1, s, s.start)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k = 0
                n0 = 1
                dir = 1
                while bool and k < 4:
                    n0 += dir
                    n0 = n0 % 4
                    if n0 == 0:
                        n0 = 4
                    s_new, b_corner = self.next_edge(n0, 1, s, corner)
                    Ls_edge.append(s_new)
                    bool, corner = b_corner
                    k += 1

            if s.start.y == 500:
                s_new, b_corner = self.next_edge(2, 0, s, s.start)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k = 0
                n0 = 2
                dir = -1
                while bool and k < 4:
                    n0 += dir
```

```
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

            s_new, b_corner = self.next_edge(2, 1, s, s.start)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 2
            dir = 1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

        if s.start.x == 0:
            s_new, b_corner = self.next_edge(3, 0, s, s.start)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 3
            dir = -1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

            s_new, b_corner = self.next_edge(3, 1, s, s.start)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 3
            dir = 1
```

```
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

        if s.start.y == 0:
            s_new, b_corner = self.next_edge(4, 0, s, s.start)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 4
            dir = -1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

            s_new, b_corner = self.next_edge(4, 1, s, s.start)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 4
            dir = 1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

    # on fait pareil avec les s.end

        if s.end.x == 500:
            s_new, b_corner = self.next_edge(1, 0, s, s.end)
```

```
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 1
            dir = -1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

            s_new, b_corner = self.next_edge(1, 1, s, s.end)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 1
            dir = 1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

        if s.end.y == 500:
            s_new, b_corner = self.next_edge(2, 0, s, s.end)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 2
            dir = -1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1
```

```
            s_new, b_corner = self.next_edge(2, 1, s, s.end)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 2
            dir = 1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

        if s.end.x == 0:
            s_new, b_corner = self.next_edge(3, 0, s, s.end)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 3
            dir = -1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 0, s, corner)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k += 1

            s_new, b_corner = self.next_edge(3, 1, s, s.end)
            Ls_edge.append(s_new)
            bool, corner = b_corner
            k = 0
            n0 = 3
            dir = 1
            while bool and k < 4:
                n0 += dir
                n0 = n0 % 4
                if n0 == 0:
                    n0 = 4
                s_new, b_corner = self.next_edge(n0, 1, s, corner)
                Ls_edge.append(s_new)
```

```
                    bool, corner = b_corner
                    k += 1

            if s.end.y == 0:
                s_new, b_corner = self.next_edge(4, 0, s, s.end)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k = 0
                n0 = 4
                dir = -1
                while bool and k < 4:
                    n0 += dir
                    n0 = n0 % 4
                    if n0 == 0:
                        n0 = 4
                    s_new, b_corner = self.next_edge(n0, 0, s, corner)
                    Ls_edge.append(s_new)
                    bool, corner = b_corner
                    k += 1

                s_new, b_corner = self.next_edge(4, 1, s, s.end)
                Ls_edge.append(s_new)
                bool, corner = b_corner
                k = 0
                n0 = 4
                dir = 1
                while bool and k < 4:
                    n0 += dir
                    n0 = n0 % 4
                    if n0 == 0:
                        n0 = 4
                    s_new, b_corner = self.next_edge(n0, 1, s, corner)
                    Ls_edge.append(s_new)
                    bool, corner = b_corner
                    k += 1
        for s1 in Ls_edge:
            for s2 in Ls_edge:
                if s1 != s2 and [int(s1.start.x), int(s1.start.y), int(s1.end.
↪x), int(s1.end.y)] == [int(s2.start.x), int(s2.start.y), int(s2.end.x),␣
↪int(s2.end.y)]:
                    Ls_edge.remove(s2)
                elif s1 != s2 and [int(s1.start.x), int(s1.start.y), int(s1.end.
↪x), int(s1.end.y)] == [int(s2.end.x), int(s2.end.y), int(s2.start.x), int(s2.
↪start.y)]:
                    Ls_edge.remove(s2)
                elif [int(s2.start.x), int(s2.start.y)] == [int(s2.end.x),␣
↪int(s2.end.y)]:
```

```
                    Ls_edge.remove(s2)
                elif s1 != s2 and self.is_in_large(s1, s2):
                    Ls_edge.remove(s2)

        return Ls_edge
```

On a ici fait appel à la fonction is_in_large qui nettoie en partie la sortie en enlevant les segments au bord qui comportent d'autres segments en leur intérieur ceux-ci étant des erreurs du processus.

[ ]:
```
    def is_in_large(self, s1, s2):
        if s1.start.x == s1.end.x and s2.start.x == s2.end.x and s1.start.x ==␣
↪s2.start.x:
            if s1.start.y < s1.end.y:
                if (s1.start.y >= s2.start.y and s1.end.y <= s2.end.y) or (s1.
↪start.y >= s2.end.y and s1.end.y <= s2.start.y):
                    return True
            if s1.start.y >= s1.end.y:
                if (s1.end.y >= s2.start.y and s1.start.y <= s2.end.y) or (s1.
↪end.y >= s2.end.y and s1.start.y <= s2.start.y):
                    return True
        elif s1.start.y == s1.end.y and s2.start.y == s2.end.y and s1.start.y␣
↪== s2.start.y:
            if s1.start.x <= s1.end.x:
                if (s1.start.x >= s2.start.x and s1.end.x <= s2.end.x) or (s1.
↪start.x >= s2.end.x and s1.end.x <= s2.start.x):
                    return True
            if s1.start.x > s1.end.x:
                if (s1.end.x >= s2.start.x and s1.start.x <= s2.end.x) or (s1.
↪end.x >= s2.end.x and s1.start.x <= s2.start.x):
                    return True
        else:
            return False
```

La liste des segments aux bords est finies on ajoute une fonction clean_output afin de finir le nettoyage de la liste des segments pour éviter tous les problèmes.

[ ]:
```
    def clean_output(self):
        for s1 in self.output:
            if s1.end is None:
                self.output.remove(s1)

        for s1 in self.output:
            for s2 in self.output:
                if s1 != s2 and [s1.start.x, s1.start.y, s1.end.x, s1.end.y] ==␣
↪[s2.start.x, s2.start.y, s2.end.x, s2.end.y]:
                    self.output.remove(s2)
```

On a donc fini pour la méthode des segments ici on à accès à l'ensemble des segments qui composent

les cellules du diagramme et on a créé les outils permettant d'en déduire le score de chaque joueur.

3.2 2.2 La méthode des polygones

Grâce au travail préliminaire fait sur les segments aux bords du Canva il était maintenant plus simple d'accéder aux coordonnées des sommets des cellules de Voronoï. Pour former les cellules de Voronoï sur le Canva nous avons déjà parcouru les germes de Voronoï pour leur associer les segments qui composent leur cellules via la fonction *upd_pol*. Une fois ce travail fait pour un point la liste de coordonnées des sommets est stocké dans l'attribut *polygons* de *Player*.

[ ]:
```
def upd_pol(self, points):
        for p in points:
            pol = []
            for s in self.output:
                if s.p1 is not None and (s.p1 == p or s.p2 == p):
                    #point0 = (s.start.x, s.start.y)
                    #point1 = (s.end.x, s.end.y)
                    point0 = (int(s.start.x), int(s.start.y))
                    point1 = (int(s.end.x), int(s.end.y))
                    if point0 not in pol:
                        pol.append(point0)
                    if point1 not in pol:
                        pol.append(point1)
            if len(pol) > 0:
                p.player.add_pol(pol)
        sort(self.player.polygons)
        sort(self.bot.polygons)
```

Une fois les coordonnées des sommets récupérés, il suffit de calculer leur aire via la formule "Shoelace formula".

[ ]:
```
#Fonction qui calcule l'aire d'un polygone dont la liste des coordonnées des␣
↪sommets est donnée en entrée
def polygon_area(coords):
    #Coordonnées des sommets
    x = [point[0] for point in coords]
    y = [point[1] for point in coords]
    # Décalage des coordonnées
    x_ = x - np.mean(x)
    y_ = y - np.mean(y)
    # Calcul de l'aire grâce à la "Shoelace formula"
    correction = x_[-1] * y_[0] - y_[-1] * x_[0]
    main_area = np.dot(x_[:-1], y_[1:]) - np.dot(y_[:-1], x_[1:])
    return 0.5 * np.abs(main_area + correction)
```

[ ]:
```
#Application de polygon_area à une liste de polygon
def area(list_coords):
    return np.sum([polygon_area(coords) for coords in list_coords])
```

Il est important que pour tracer les polygones désirés ces fonctions doivent avoir en entrée la liste des cordonnées des sommets **triée** pour éviter que le polygone tracé soit croisé. C'est la fonction *sort* qui nous garantit que les sommets soit ordonnés dans l'ordre horaire en partant du centre du polygone.

[18]:
```
Image('https://i.ibb.co/BTHkd4X/pentagone-etoile.png')
```

[18]:

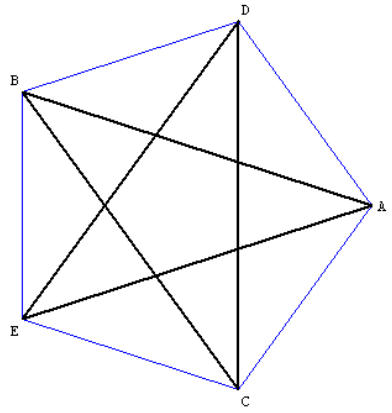

Figure 11 - Exemple de polygone croisé (qu'on ne veut pas tracer) et son envelopper convexe (qu'on veut tracer)

[ ]:
```
#Fonction qui trie la liste des sommets des polygons selon l'ordre horaire␣
 ↪autour du centre du polygone
def sort(pol):
    for single_pol in pol:
        cent=(sum([p[0] for p in single_pol])/len(single_pol),sum([p[1] for p␣
 ↪in single_pol])/len(single_pol))
        single_pol.sort(key=lambda p: math.atan2(p[1]-cent[1],p[0]-cent[0]))
    return pol
```

4 2)Interface Joueur

Pour implémenter l'interface graphique nous avons choisi de travailler avec le module tkinter. Grâce au travail réalisé précédemment sur la structure du diagramme de Voronoï l'implémentation est facilitée. Dans l'interface graphique, la couleur bleue est utilisée pour l'ordinateur tandis que la couleur rouge est utilisée pour le joueur. Dans le cas du mode 2 joueurs le joueur 2 prend la couleur bleu.

4.1 2.1) Fonctionnalités présentes sur le jeu.

Dans notre implémentation plusieurs fonctionnalités sont présentes. Les scores des joueurs sont calculés grâce à la classe Voronoï et sont affichés sur l'interface graphique en temps réels. Un bouton *Mode de jeu 2 joueurs* permet de jouer à 2 joueurs bien que cela ne soit pas l'objectif

principal du projet. Plusieurs boutons de stratégie de l'ordinateur sont présents, un click sur l'un de ces boutons donne une valeur à *self.strategy* qui change la fonction de placement de point de l'ordinateur dans la fonction *onDoubleClick*. Un bouton *Rejouer* permet de réinitialiser le jeu en effaçant le diagramme et réinitialisant les scores à 0. Une fonction *check_winner* permet d'afficher le vainqueur du jeu au tour 5 en comparant les scores des deux joueurs.

[ ]:
```
class MainWindow:
    # Rayon des points affichés sur le tkinter
    RADIUS = 3

    #Variable qui permet de vérouiller la fenêtre tkinter pour tracer le␣
↪diagramme
    LOCK_FLAG = False

    def __init__(self, master):



        self.master = master
        self.master.title("Voronoi")

        self.frmMain = tk.Frame(self.master, relief=tk.RAISED, borderwidth=1)
        self.frmMain.pack(fill=tk.BOTH, expand=1)

        self.w = tk.Canvas(self.frmMain, width=500, height=500)
        self.w.config(background='white')
        self.w.bind('<Double-1>', self.onDoubleClick)

        self.w.pack()

        self.frmButton = tk.Frame(self.master)
        self.frmButton.pack()

        #Bouton de choix du mode jeu
        self.btn2users = tk.Button(self.frmButton, text='Mode de jeu 2␣
↪joueurs', width=25, command=self.mode_2_users)
        self.btn2users.pack(side=tk.LEFT)

        #Bouton de choix de la stratégie de l'ordinateur#
        self.btnGreedyBot = tk.Button(self.frmButton, text='Stratégie Bonne␣
↪Pioche', width=25, command=self.BonnePiocheBot)
        self.btnGreedyBot.pack(side=tk.LEFT)

        self.btnAntiGagnantBot = tk.Button(self.frmButton, text='Stratégie Anti␣
↪Gagnant', width=25, command=self.AntiGagnantBot)
        self.btnAntiGagnantBot.pack(side=tk.LEFT)
```

```
        self.btnDBCBot = tk.Button(self.frmButton, text='Stratégie DBC',␣
↪width=25, command=self.DBCBot)
        self.btnDBCBot.pack(side=tk.LEFT)

        self.btnrandomBot = tk.Button(self.frmButton, text='Placement aléatoire␣
↪(Par défaut)', width=25, command=self.randomBot)
        self.btnrandomBot.pack(side=tk.LEFT)


        #Bouton de reset du jeu#
        self.btnClear = tk.Button(self.frmButton, text='Rejouer', width=25,␣
↪command=self.onClickClear)
        self.btnClear.pack(side=tk.LEFT)


        #Affichage des scores du joueurs et du bot#
        self.score_user = 0
        self.score_user_variable = tk.StringVar(self.master, f'Score Joueur:␣
↪{self.score_user}')
        self.score_user_lbl = tk.Label(self.master, textvariable=self.
↪score_user_variable)
        self.score_user_lbl.pack()

        self.score_bot = 0
        self.score_bot_variable = tk.StringVar(self.master, f'Score Bot: {self.
↪score_bot}')
        self.score_bot_lbl = tk.Label(self.master, textvariable=self.
↪score_bot_variable)
        self.score_bot_lbl.pack()


        #Variable de compteur de tour#
        self.count = 0

        #Variable de choix de stratégie#
        self.strategy = 0


        #Variable mode de jeu 1 ou 2 joueurs
        self.game_mode = 0



    def mode_2_users(self):
        self.game_mode = 1
    # Pour le choix des stratégies, le bouton de chaque stratégie affecte une
```

```
    # valeur à la variable strategy qui vient changer la fonction de placement
    # de point du bot utilisé dans onDoubleClick
    def randomBot(self):
        self.strategy = 0

    def BonnePiocheBot(self):
        self.strategy = 1

    def AntiGagnantBot(self):
        self.strategy = 2

    def DBCBot(self):
        self.strategy = 3

    #Définition du bouton Clear : Clear reset le jeu#
    def onClickClear(self):
        self.LOCK_FLAG = False
        self.w.delete(tk.ALL)
        self.score_user = 0
        self.score_bot = 0
        self.score_user_variable.set(f'Score Joueur: {self.score_user}')
        self.score_bot_variable.set(f'Score Bot: {self.score_bot}')
        self.count = 0
        self.strategy = 0
        self.game_mode = 0
```

4.2 2.2) Principe de création du diagramme sur la fenêtre de jeu

La commance *self.w.find_all* nous permet de récupérer les coordonnées des points placés sur la fenêtre de jeu pour ensuite créer le diagramme de Voronoï correspondant à ces germes via la fonction *get_vp*.

[ ]:
```
    def get_vp(self):
        pObj = self.w.find_all()
        points = []
        for p in pObj:
            if self.w.itemcget(p, "fill") == "red":
                coord = self.w.coords(p)
                points.append((coord[0]+self.RADIUS, coord[1]+self.RADIUS,1))
            if self.w.itemcget(p, "fill") == "blue" or self.w.itemcget(p,␣
↪"fill") == "yellow":
                coord = self.w.coords(p)
                points.append((coord[0]+self.RADIUS, coord[1]+self.RADIUS,0))

        vp = Voronoi(points)
        vp.process()
        vp.act_score2()
```

```
        return vp
```

La fonction centrale du jeu est la fonction onDoubleClick qui crée un point sur la fenêtre à l'endroit où le joueur double clique et place le coup joué par l'ordinateur. La fonction réalise immédiatement l'actualisation des scores qui sont affichés en bas de la fenêtre. Nous récupérons alors les coordonnées des débuts et fin de segments du diagramme et traçons ces segments sur le diagramme grâce a *drawlinesonCanvas*.

Comme le diagramme de Voronoï du tour précédent est encore affiché sur la fenêtre il est impératif que cette fonction efface les lignes et les polygones du diagramme précédent. C'est pourquoi nous utilisons *self.w.delete("lines")* qui supprimes les anciennes lignes et *self.w.delete("poly")* qui supprime les anciens polygones.

[ ]:
```
    def onDoubleClick(self, event):

        # On vérifie d'abord le mode de jeu

        # Mode de jeu solo
        if self.game_mode == 0:

            # On vérifie si le jeu doit s'arreter
            if self.count == 5:
                self.check_winner()

            else:
                if not self.LOCK_FLAG:
                    self.w.create_oval(event.x-self.RADIUS, event.y-self.RADIUS,
                                       event.x+self.RADIUS, event.y+self.
↪RADIUS, fill="red")

                    self.LOCK_FLAG = True
                    # On efface les lignes du diagramme précédent

                    self.w.delete("lines")
                    self.w.delete("poly")

                    #Selection de la stratégie du bot #
                    if self.strategy == 0:
                        self.random_place()

                    elif self.strategy == 1:
                        self.BonnePioche_place()

                    elif self.strategy == 2:
                        self.AntiGagnant_place()

                    elif self.strategy == 3:
                        self.DBC_place()
```

```
                # On calcule le diagramme de Voronoi
                vp = self.get_vp()
                lines = vp.get_output()

                #Actualisation du score du joueur et du bot #

                self.score_user = int(
                    1000*vp.player.score/(vp.bot.score + vp.player.score+1))/10
                self.score_bot = int(
                    1000*vp.bot.score/(vp.bot.score + vp.player.score+1))/10

                self.score_user_variable.set(
                    f'Score Joueur: {self.score_user}%')
                self.score_bot_variable.set(f'Score Bot: {self.score_bot}%')

                # Tracer du diagramme de Voronoï
                self.drawPolygonOnCanvas(vp)
                self.drawLinesOnCanvas(lines)

                # Incrémentation du compteur de tour
                self.count += 1

                self.LOCK_FLAG = False

                if self.count == 5:
                    self.check_winner()

        # Mode de jeu 2 joueurs
        else:

            if self.count == 10:
                self.check_winner()

            # On décide de la couleur du point en fonction du tour
            if self.count % 2 == 0 and not self.LOCK_FLAG:
                self.w.create_oval(event.x-self.RADIUS, event.y-self.RADIUS,
                                   event.x+self.RADIUS, event.y+self.RADIUS,␣
↪fill="red")

            if self.count % 2 == 1 and not self.LOCK_FLAG:
                self.w.create_oval(event.x-self.RADIUS, event.y-self.RADIUS,
                                   event.x+self.RADIUS, event.y+self.RADIUS,␣
↪fill="blue")

            self.LOCK_FLAG = True
```

```
            # On efface les lignes du diagramme précédent

            self.w.delete("lines")
            self.w.delete("poly")

            # On calcule le diagramme de Voronoi
            vp = self.get_vp()
            lines = vp.get_output()

            # Actualisation du score des joueurs
            self.score_user = int(1000*vp.player.score /
                                  (vp.bot.score + vp.player.score+1))/10
            self.score_bot = int(

                1000*vp.bot.score/(vp.bot.score + vp.player.score+1))/10

            self.score_user_variable.set(f'Score Joueur 1: {self.score_user}%')
            self.score_bot_variable.set(f'Score Joueur 2: {self.score_bot}%')

            # Tracer du diagramme de Voronoï
            self.drawPolygonOnCanvas(vp)
            self.drawLinesOnCanvas(lines)

            # Incrémentation du compteur de tour
            self.count += 1

            self.LOCK_FLAG = False

            if self.count == 10:
                self.check_winner()

    def drawLinesOnCanvas(self, lines):

        for l in lines:
            self.w.create_line(l[0], l[1], l[2], l[3], width=2, tags="lines")

    def drawPolygonOnCanvas(self, vp):

        for single_pol in vp.player.polygons:
            pol_trace = list(itertools.chain(single_pol))
            self.w.create_polygon(pol_trace, fill="red",
                                  stipple='gray50', tags="poly")

        for single_pol in vp.bot.polygons:
            pol_trace = list(itertools.chain(single_pol))
            self.w.create_polygon(pol_trace, fill="blue",
                                  stipple='gray50', tags="poly")
```

[6]:
```
Image('https://i.ibb.co/KNWk5Xk/random.png', embed=False)
```

[6]:
```
<IPython.core.display.Image object>
```

Figure 12 - Exemple d'une partie sur notre jeu de Voronoï

5 3)Stratégie

5.1 Introduction

Nous avons implémenter certaines des stratégies présentes sur le site Interstices.info . Nous avons aussi ajouter une nouvelle stratégie dont l'idée originelle est tirée d'un article sur les stratégies au jeu de Voronoï 2D

5.2 3.1) Stratégie aléatoire

La première "stratégie" implémentée est un placement aléatoire des points par l'ordinateur.

[ ]:
```
    #Placement du point aléatoirement #

    def random_place(self):
        # points est la liste de points déjà sur le plan on ajoute juste le␣
↪point que place le bot
        x = r.random()*500
        y = r.random()*500
        self.w.create_oval(x-self.RADIUS, y-self.RADIUS, x +
                           self.RADIUS, y+self.RADIUS, fill="blue")
```

La Figure 12 est utilisée avec une stratégie aléatoire.

5.3 3.2) Stratégie Bonne Pioche

Cette stratégie est un rafinement de la première, l'ordinateur joue 20 coups aléatoires et sélectionne le coup qui maximise son score. Pour l'implémenter nous générons le diagramme de Voronoï grâce à la fonction *get_vp* dans les 20 coups possibles et sélectionnons le coup qui maximise le score de l'ordinateur.

[ ]:
```
#Placement du point selon la stratégie bonne pioche#

    def BonnePioche_place(self):

        xy = [(r.random()*500, r.random()*500) for i in range(20)]

        self.w.create_oval(xy[0][0]-self.RADIUS, xy[0][1]-self.RADIUS, xy[0]
                           [0]+self.RADIUS, xy[0][1]+self.RADIUS,␣
↪fill="yellow", tags='train')
        vp = self.get_vp()
```

```
        maxi = vp.bot.score
        (x_play, y_play) = (0, 0)
        self.w.delete("train")

        # Boucle de recherche du max
        for (x, y) in xy:
            self.w.create_oval(x-self.RADIUS, y-self.RADIUS, x +
                               self.RADIUS, y+self.RADIUS, fill="yellow",␣
↪tags='train')
            vp = self.get_vp()
            if vp.bot.score > maxi:
                maxi = vp.bot.score
                (x_play, y_play) = (x, y)
            self.w.delete("train")
        self.w.create_oval(x_play-self.RADIUS, y_play-self.RADIUS,
                           x_play+self.RADIUS, y_play+self.RADIUS, fill="blue")
```

[11]:
```
Image("https://i.ibb.co/6yBrWr4/Bonne-pioche.png")
```

[11]:

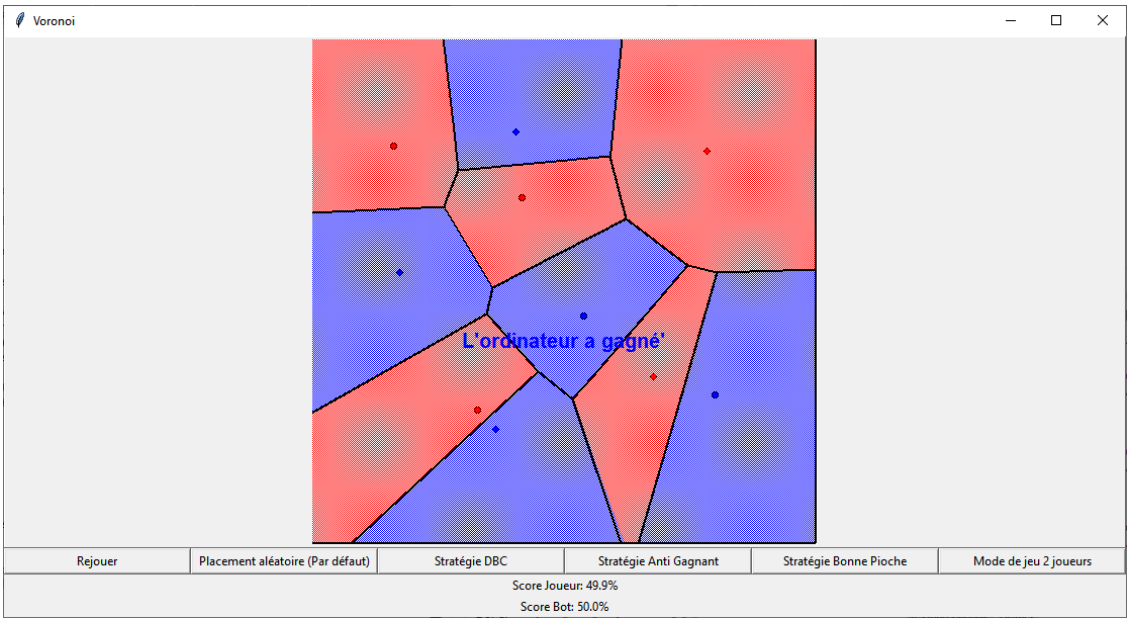

Figure 13 - Exemple de partie avec la stratégie Bonne Pioche

5.4 3.3) Stratégie Anti Gagnant

La stratégie Anti Gagnant divise la plus grosse celulle de l'adversaire en occupant son centre. On recherche donc la cellule d'aire maximale et on calcule son centre pour décider du coup à jouer. Pour le premier coup la liste de polygone du joueur est vide, cela ne pose pas de problème car AntiGagnant joue toujours le point central du jeu en premier coup.

[ ]:
```
def AntiGagnant_place(self):
        if self.count == 0:
            self.w.create_oval(250-self.RADIUS, 250-self.RADIUS,
                               250+self.RADIUS, 250+self.RADIUS, fill="blue")
        else:
            vp = self.get_vp()
            list_coords = vp.player.polygons
            list_aire = [polygon_area(coords) for coords in list_coords]
            i = list_aire.index(max(list_aire))
            centre_pol_i = (sum([p[0] for p in list_coords[i]])/
↪len(list_coords[i]),
                            sum([p[1] for p in list_coords[i]])/
↪len(list_coords[i]))

            (x_play, y_play) = centre_pol_i
            self.w.create_oval(x_play-self.RADIUS, y_play-self.RADIUS,
                               x_play+self.RADIUS, y_play+self.RADIUS,␣
↪fill="blue")
```

[12]:
```
Image("https://i.ibb.co/5B8c9LG/Anti-Gagnant.png")
```

[12]:

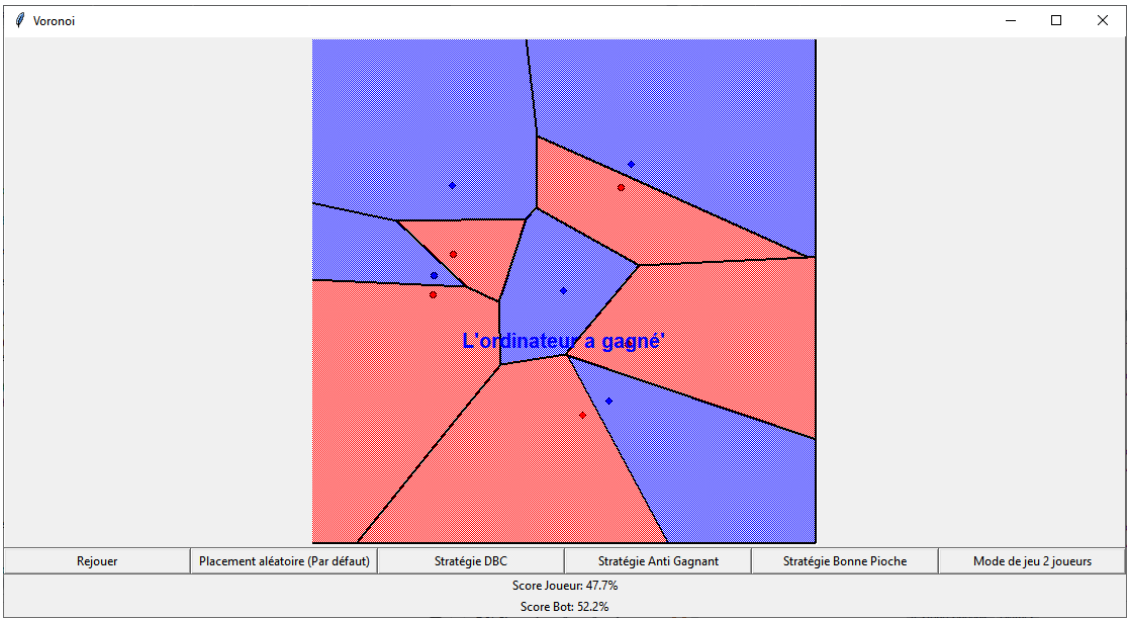

Figure 14 - Exemple de partie avec la stratégie Anti-Gagnant

5.5 3.3) Stratégie Defensive Balanced Cells modifiée

5.5.1 3.3.1) Stratégie Defensive Balanced Cells

La stratégie Defensive Balanced Cells (non modifiée) (DBC) est une stratégie assez surprenante, le placement des points est prédéfini et ne prend pas en compte les coups joués par l'adversaire. Cette stratégie repose sur une notion centrale des diagrammes de Voronoï : l'équilibre. Le centre de gravité est défini comme ceci : une droite qui passe par un centre de gravité d'une cellule coupe la cellule en deux parties d'aires égales. Une cellule dont le germe est situé sur le centre de gravité est dite équilibrée. Un diagramme équilibré est alors un diagramme dont toutes les cellules sont équilibrées.

[16]:
```
Image('https://i.ibb.co/GdFVPtH/equilibr.png', embed=False)
```

[16]:
```
<IPython.core.display.Image object>
```

Figure 15 - Exemple de diagramme de Voronoï équilibré pour 5 germes

L'une des attaques possibles dans le jeu de Voronoï est d'occupé le centre de gravité d'une cellule, ce qui permet de garantir de voler au moins la moitié de la surface de la cellule. Pour éviter cette attaque on peut alors tenter de construire un diagramme équilibré. C'est le but de cette stratégie.

5.5.2 3.3.2) Stratégie Defensive Balanced Cells modifiée

Face à un joueur conscient que l'ordinateur joue cette stratégie l'intérêt est très limité car les coups sont prévisibles. Dans cette stratégie on joue donc autour des points qui donne un diagramme équilibré pour ajouter de l'imprévue. Cette stratégie porte aussi l'idée naturelle de vouloir uniformiser la densité de points.

[ ]:
```
#Placement du point selon la stratégie Defensive balanced cell#

    def DBC_place(self):
        # Cette liste contient les coordonnées d'un diagramme de Voronoï␣
↪équilibré à 5 points
        DBC_list = [(400, 400), (100, 400), (400, 100), (100, 100), (250, 250)]

        # Cette liste contient une version legerement modifié de la liste␣
↪précédente pour être plus imprévisible
        play_list = [(x + random.choice((-1, 1))*r.random()*25, y +
                      random.choice((-1, 1))*r.random()*25) for (x, y) in␣
↪DBC_list]

        i = self.count
        (x, y) = play_list[i]
        self.w.create_oval(x-self.RADIUS, y-self.RADIUS, x +
                           self.RADIUS, y+self.RADIUS, fill="blue")
```

[13]:
```
Image("https://i.ibb.co/v1DhGwQ/DBC.png")
```

[13]:

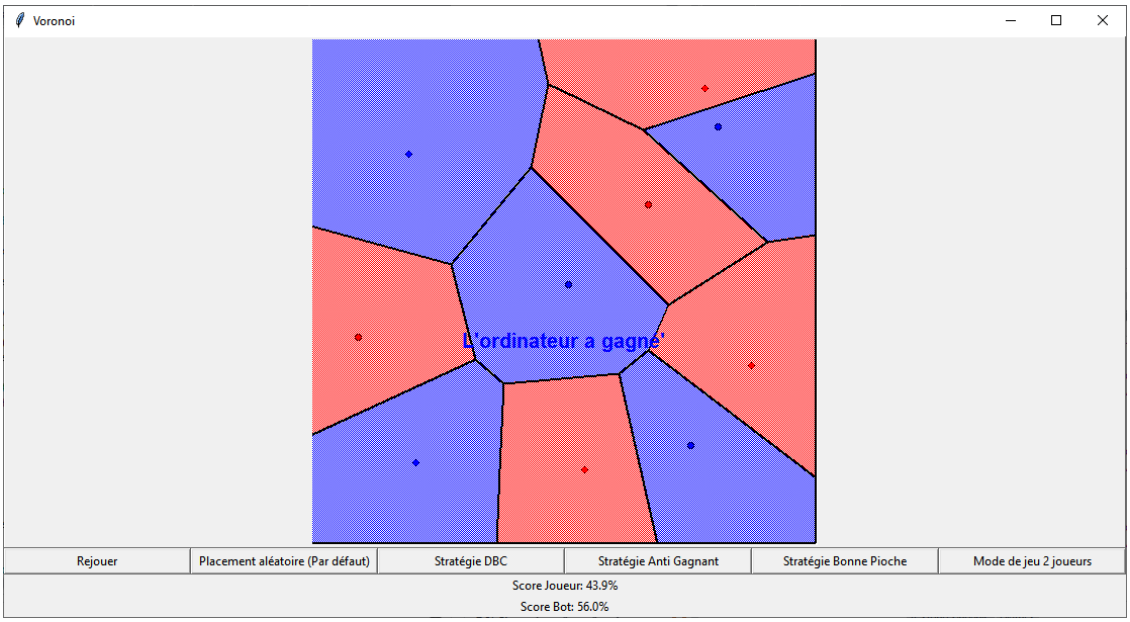

Figure 16 - Exemple de partie avec la stratégie DBC modifiée