TIPE

Détermination de trajectoires rapides et sûres pour robots

François-Xavier Courel - Candidat 19136

Toutes les images non sourcées ont été produites par l'auteur du TIPE

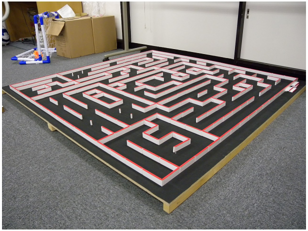
Sommaire

Sommaire

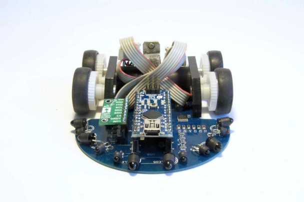
Introduction au problème

Micromouse

- Course de robots appelées "souris" dans un labyrinthe :



Source : Osamu Iwasaki

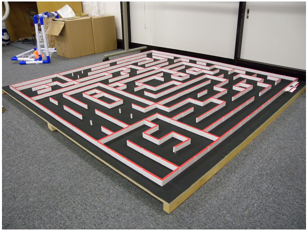


Source : alexhadik.com/work/micromouse/

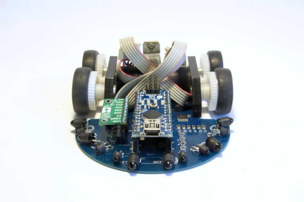
Introduction au problème

Micromouse

- Course de robots appelées "souris" dans un labyrinthe :



Source : Osamu Iwasaki



Source : alexhadik.com/work/micromouse/

- Les règles ? Quelques minutes pour :
  1. Explorer le labyrinthe, enregistrer la position des murs
  2. Calculer le chemin le plus rapide
  3. Faire plusieurs tentatives de courses

Position du problème

- **Objectif 1 :** Trouver le chemin le plus rapide

Position du problème

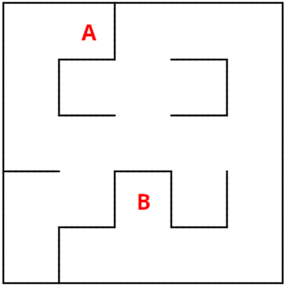
- **Objectif 1 :** Trouver le chemin le plus rapide
- **Objectif 2 :** Éviter les collisions avec les obstacles

Position du problème

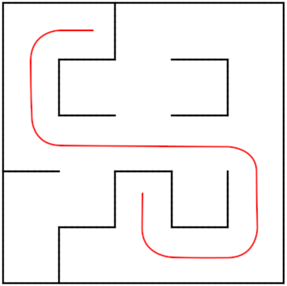
- **Objectif 1 :** Trouver le chemin le plus rapide
- **Objectif 2 :** Éviter les collisions avec les obstacles
- **Objectif 3 :** Temps de calcul le plus court possible

Position du problème

- **Objectif 1 :** Trouver le chemin le plus rapide
- **Objectif 2 :** Éviter les collisions avec les obstacles
- **Objectif 3 :** Temps de calcul le plus court possible



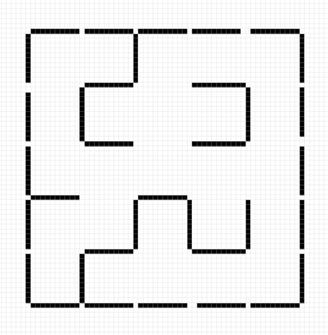
$\longmapsto$



Résolution naïve

Méthode

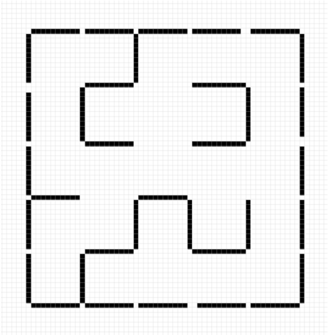
1. Construire la carte en discrétisant les coordonnées

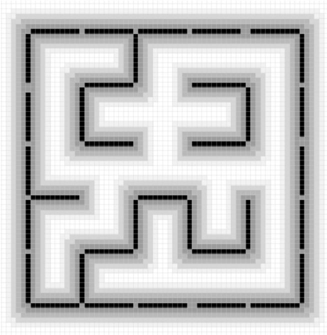


Résolution naïve

Méthode

1. Construire la carte en discrétisant les coordonnées
2. Propager depuis les obstacles la distance de sécurité

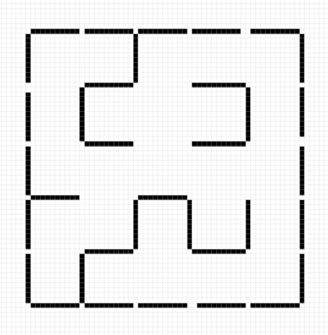


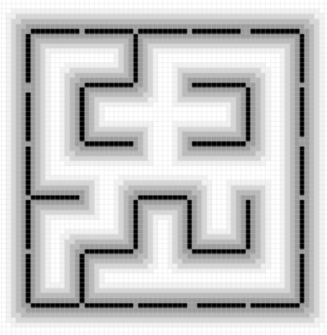


Résolution naïve

Méthode

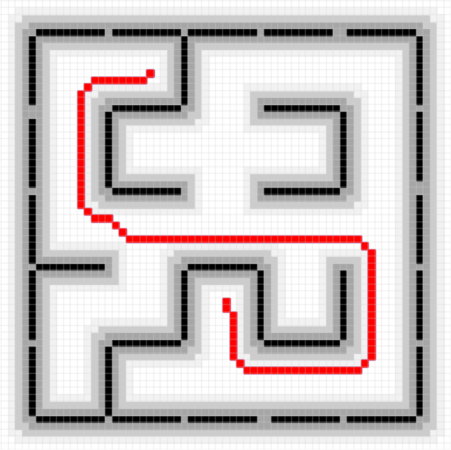
1. Construire la carte en discrétisant les coordonnées
2. Propager depuis les obstacles la distance de sécurité
3. Calculer un chemin via un algo classique (A* ou Dijkstra)





Résolution naïve

Un premier résultat !



- Cet algorithme tend à longer les murs
- Complexité :

1. $O(N)$ où N est le nombre de cellules
2. $O(N\log(N))$
3. $O(N\log(N))$

Et $N \propto \frac{1}{p^2}$ ou p représente la précision

Diagramme de Voronoï

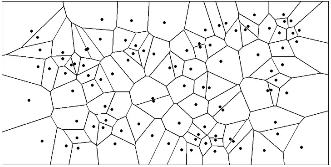
Définition

On se place dans $\mathbb{R}^2$. Soit $S := \{s_1, ..., s_n\}$ un ensemble *fini* de n points du plan, qu'on appellera *sites*.

Définition (Cellule de Voronoï)

Soit $s \in S$. On appelle cellule de Voronoï de s la région du plan consitituée de tous les points dont s est le site le plus proche :

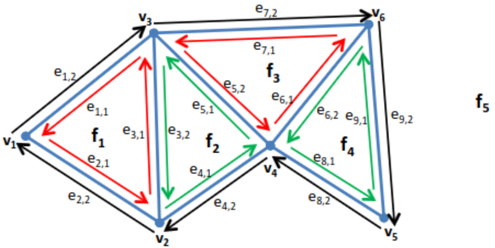
$$\mathrm{Vor}(s) = \{p \in \mathbb{R}^2 : \forall s' \in S, \|p - s\| \leqslant \|p - s'\|\}$$



Source : towardsdatascience.com

Représentation

DCEL (Liste d'arêtes doublement chaînées)

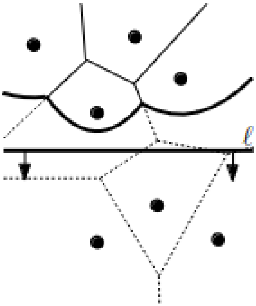


Source : Simon Fraser University

- Représentation de graphes planaires
- Permet de retrouver à partir d'une arête les deux sites des deux faces adjacentes

Algorithme de Fortune

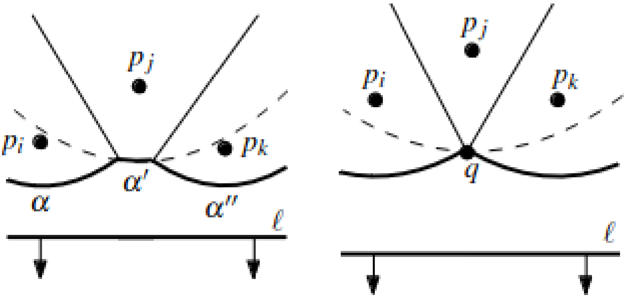
- Algorithme avec ligne de balayage (*sweep line*) et ligne de front (*beach line*)
- *Site events*
- *Circle events*



Source : Computational Geometry

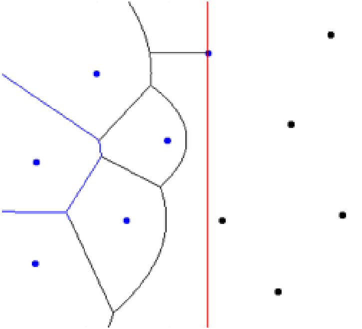
Algorithme de Fortune

Circle events



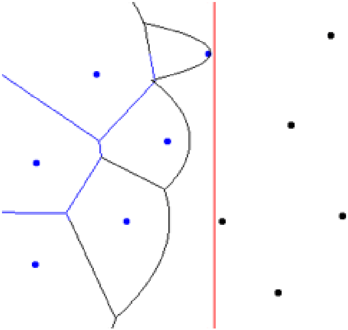
Source : Computational Geometry

Un exemple



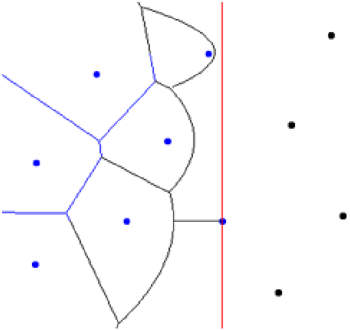
Source : Wikipedia

Un exemple



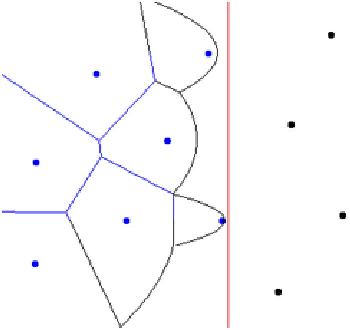
Source : Wikipedia

Un exemple



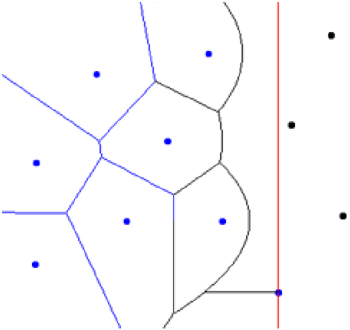
Source : Wikipedia

Un exemple



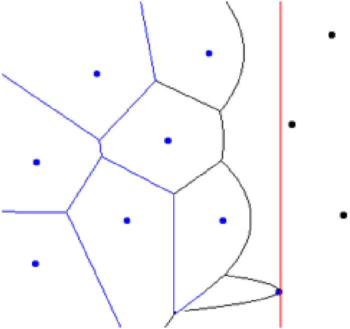
Source : Wikipedia

Un exemple



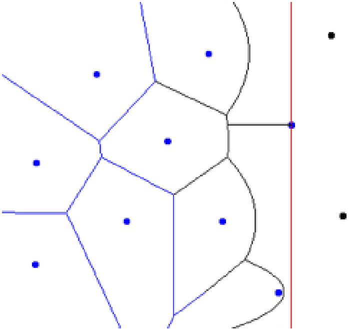
Source : Wikipedia

Un exemple



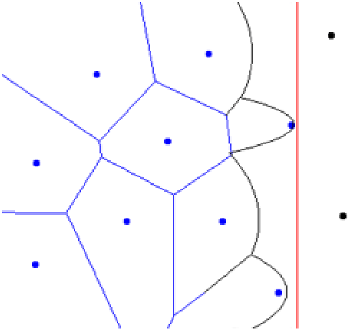
Source : Wikipedia

Un exemple



Source : Wikipedia

Un exemple



Source : Wikipedia

Algorithme de Fortune

$T, A, D \leftarrow$ Tas($s_1, ..., s_n$), Arbre(), DCEL()
Tant que T n'est pas vide, **faire** :
  Retirer de T l'élément minimal e
  Si c'est un *site event* **alors**
    TRAITEMENTSITEEVENT(e)
  Sinon
    a $\leftarrow$ Recherche dans l'arbre A de l'arc disparaissant.
    TRAITEMENTCIRCLEEVENT(a)
  Fin si
Fin tant que

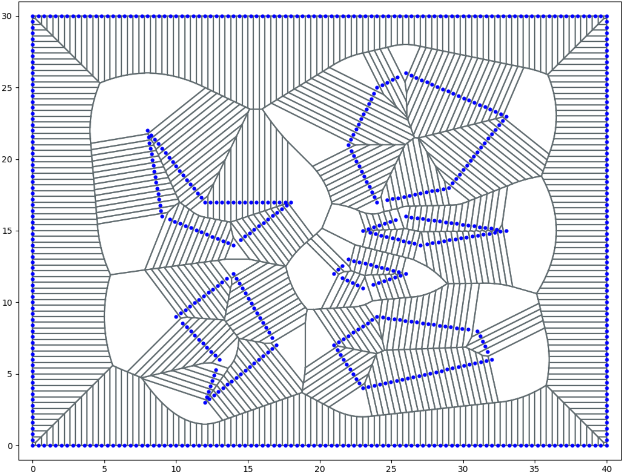
Algorithme de Fortune

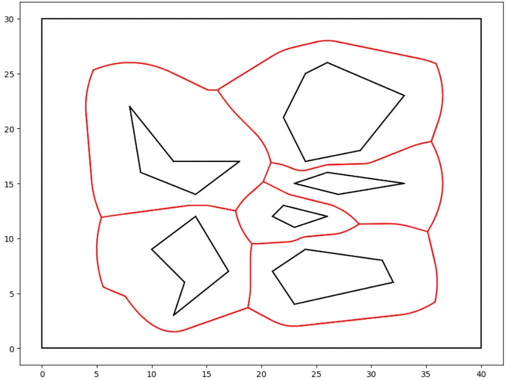
$T, A, D \leftarrow$ Tas($s_1, ..., s_n$), Arbre(), DCEL()
Tant que T n'est pas vide, **faire** :
    Retirer de T l'élément minimal e
    Si c'est un *site event* **alors**
        TraitementSiteEvent(e)
    Sinon
        a $\leftarrow$ Recherche dans l'arbre A de l'arc disparaissant.
        TraitementCircleEvent(a)
    Fin si
Fin tant que

Théorème

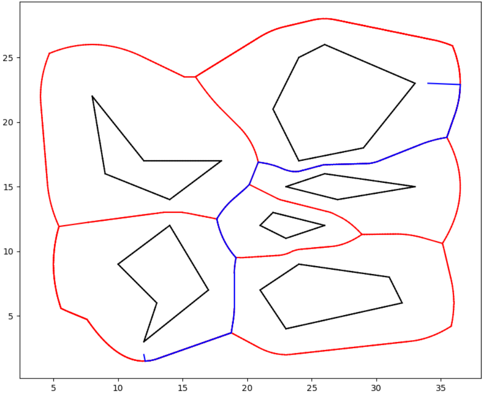
L'algorithme de Fortune a une complexité temporelle en $O(n \log n)$ et spatiale en $O(n)$. Cette complexité est optimale.

Mise en oeuvre



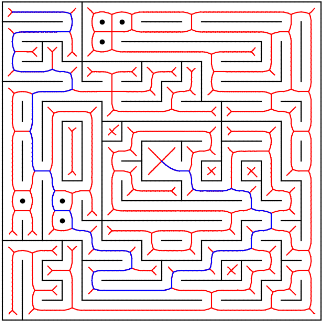


1. Récupérer les arêtes du diagramme de Voronoï
2. Conserver uniquement les arêtes "centrales"
3. Simplifier le graphe



1. Chercher les points les plus proches du départ / arrivée
2. Chercher le chemin dans le graphe simplifié
3. Reconstituer tous les points intermédiaires

Observations de cette approche

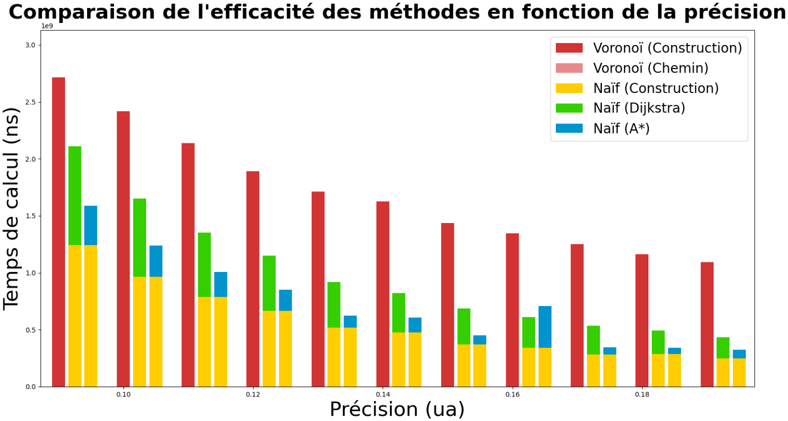


- Complexité similaire dans le pire cas :
  $O\left(\frac{1}{p^2}\log\frac{1}{p}\right)$
- Ne nécessite pas de fixer de marge à l'avance

- Une fois la carte précalculée, la recherche de chemin est instantanée
- Les chemins sont plus lisses, virages anticipés
- Les chemins évitent mieux les obstacles

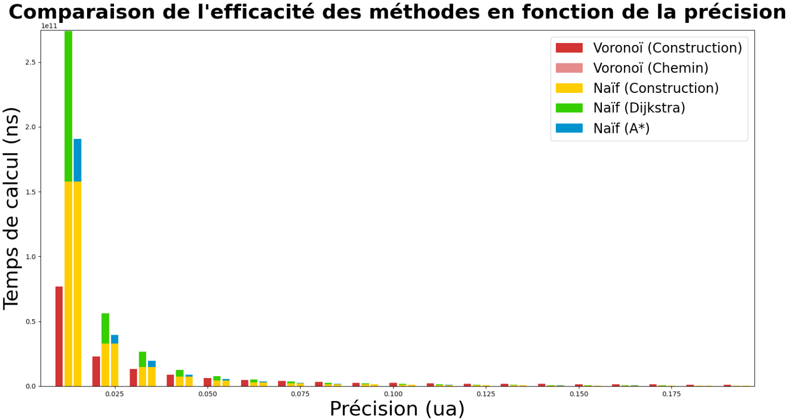
Comparaison de l'efficacité

A basse / moyenne précision

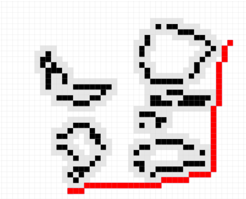
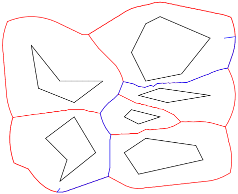
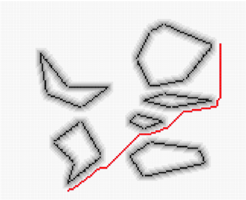
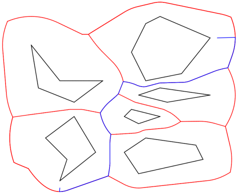


Comparaison de l'efficacité

Mais quand on regarde à haute précision...



Comparaison de la qualité

-	Naïf	Voronoï
Basse résolution		
Haute résolution		

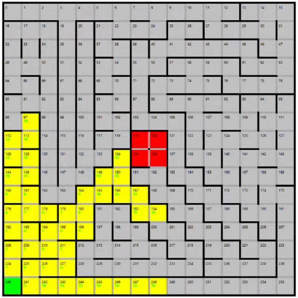
Conclusion

Diagramme de Voronoï

- Plus lent en général pour le précalcul mais plus résistant aux grandes précisions
- Résultat satisfaisant même à basse précision !
- Chemin instantané
- Trajectoires plus satisfaisantes
- Plus adaptatif à la géométrie du problème

Méthode naïve

- Rapide pour de faibles précisions, mais le résultat n'est pas toujours satisfaisant dans ce cas
- Très dépendant de la marge voulue par rapport aux obstacles
- Convient pour la géométrie très particulière de *Micromouse*



Source : `github.com/lime7git/micromouse`

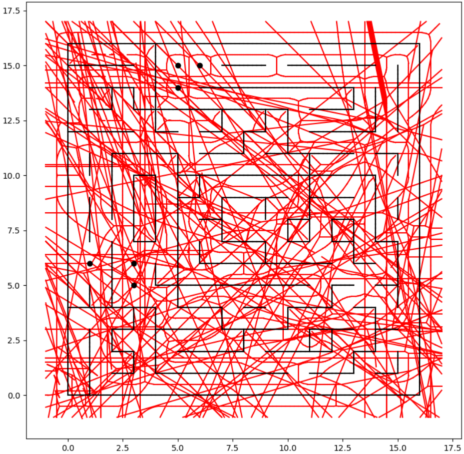
- Algorithme de floodfill (parcours en largeur)
- Les chemins sont tracés pour suivre les milieux des cases
- Pour les géométries très particulières telles que dans le cas des labyrinthes *Micromouse*.

Preuve de la complexité de l'algorithme de Fortune.

Il y a n sites, et un nombre d'évènements cercles en $O(n)$. Le traitement d'un évènement est en $O(\log n)$ (Ajout, retrait, recherches dans des tas et arbres équilibrés). D'où une complexité en $O(n \log n)$.

Optimalité : Un diagramme de Voronoï permet de trier n réels, la complexité en $O(n \log n)$ est donc optimale. $\square$

Quelques bizarreries



```python
def a_star(self, heuristique: Callable, depart: tuple, arrivee: tuple):

    queue = PriorityQueue()
    queue.put((0, (i_dep, j_dep))) # Initialisation de la file de priorité avec le départ

    parents = {}
    longueur_plus_court_chemin = {}
    longueur_plus_court_chemin[(i_dep, j_dep)] = 0

    while not queue.empty():
        _, point = queue.get() # On enlève le minimum
        i, j = point
        if point == (i_arr, j_arr):
            return reconstituer_chemin() # On est arrivé, on reconstruit le chemin

        for voisin in CarteNaif.voisins_points(i, j):
            if voisin not in self.matrice or
            self.dist_obstacle[voisin] < Settings.NAIF_MARGE:
                continue
            tentative_chemin = longueur_plus_court_chemin[point] + # Potentiel meilleur chemin
                Point.distance_eucli(i, j, *voisin)                # trouvé jusqu'à voisin
            if voisin not in longueur_plus_court_chemin or
            tentative_chemin < longueur_plus_court_chemin[voisin]:
                parents[voisin] = point
                longueur_plus_court_chemin[voisin] = tentative_chemin
                meilleure_heuristique = tentative_chemin +
                    heuristique(voisin, (i_arr, j_arr))
                queue.put((meilleure_heuristique, voisin))

    raise CheminNonTrouve("Chemin naïf non trouvé")
```

```python
class HalfEdge:

    def __init__(self, point_incident: Point, jumelle: "HalfEdge" = None, origine: Vertex = None):

        # Pointeur vers l'origine de l'arête
        self.origin = origine

        # Le point dont l'arête est incidente (ie. le point de la cellule de Voronoi)
        self.point_incident = point_incident

        # L'autre arête incidente sur le même point (dans le sens contraire)
        self._jumelle = None
        self.jumelle = jumelle

        # Suivant et précédent
        self.suiv = None
        self.prec = None
```

```python
@classmethod
def from_voronoi(cls, voronoi: Voronoi, carte_origine: Carte) -> 'CarteGraphee':

    sites: List[Point_V] = list(voronoi.sites)
    aretes: List[HalfEdge] = voronoi.edges
    vertices: List[Vertex] = voronoi.vertices

    def vertex_to_tuple(vertex: Vertex): ...
    def point_to_tuple(p: Point): ...
    def pointV_to_tuple(p: Point_V): ...
    def est_meme_obstacle(p1: Point_V, p2: Point_V): ...

    ...

    carte: CarteGraphee = cls(carte_origine)

    for e in aretes:
        center: Point_V = e.point_incident
        point_oppose: Point_V = e.jumelle.point_incident
        if point_oppose is not None:
            d = Point.distance_eucli(
                center.xd, center.yd, point_oppose.xd, point_oppose.yd)
            if d >= Settings.MULT_ESPACEMENT_SITES * Settings.PRECISION and not
            (carte_origine.obstacles_convexes and est_meme_obstacle(center, point_oppose)):

                carte.ajouter_arete((source := vertex_to_point[vertex_to_tuple(e.origine)]), (
                    dest := vertex_to_point[vertex_to_tuple(e.cible)]), source.distance(dest))

    return carte
```

```python
@classmethod
def from_scratch(cls, nom_fichier: str, bounding_size: Tuple, force_bound: bool) -> 'CarteGraphee':

    carte_raw: Carte = Carte.lire_carte(nom_fichier)
    (x_min, x_max), (y_min, y_max) = bounding_size
    if force_bound:
        carre = [
            Point(x_min, y_min),
            Point(x_min, y_max),
            Point(x_max, y_max),
            Point(x_max, y_min)
        ]
        carte_raw.obstacles.append(Obstacle(carre, False, True))
    carte_raw.decouper_obstacles(Settings.PRECISION) # Découpage des obstacles
    bounding_box = BoundingBox(x_min, x_max, y_min, y_max)
    v = Voronoi(bounding_box) # Création du diagramme de Voronoi
    v.creer_diagramme(points=list(carte_raw.voronoi_set()))

    carte: CarteGraphee = cls.from_voronoi(v, carte_raw) # Extraction de la carte simplifiée
    carte.simplifier_graphe()
    # carte.supprimer_culs_de_sac()
    return carte
```

```python
def creer_diagramme(self, points: list):

    self.initialiser(points) # Initialiser tous les points
    index = 0
    premier_point = None

    while not self.event_queue.empty():

        event = self.event_queue.get() # Retirer l'event de plus petite priorité
        premier_point = premier_point or event.point

        # Traitement des circle event
        if isinstance(event, CircleEvent) and event.est_valide:
            self.sweep_line = event.yd # Mise à jour sweep line
            self.traitement_circle_event(event)

        # Traitement des site event
        elif isinstance(event, SiteEvent):
            event.point.nom = index
            index += 1
            self.sweep_line = event.yd # Mise à jour sweep line
            self.traitement_site_event(event)
        else: continue

        self.event = event

    # Finalisation avec la bounding box
    self.edges = self.bounding_poly.finish_edges(
        edges=self.edges, vertices=self._vertices, points=self.sites, event_queue=self.event_queue
    )
```

```python
def traitement_site_event(self, event: SiteEvent):

    # Créer un nouvel arc
    point_i = event.point
    new_arc = Arc(origine=point_i)
    self._arcs.add(new_arc)

    # 1. Si la ligne de front est vide, on insère le point.
    if self.arbre_front is None:
        self.arbre_front = NoeudFeuille(new_arc)
        return

    # 2. Recherche de l'arc au dessus du points
    arc_node_above_point = Arbre_AVL.chercher_noeud_feuille(self.arbre_front, cle=point_i.xd, sweep_line=s
    arc_above_point = arc_node_above_point.get_value()

    # Retirer une potentielle fausse alerte
    if arc_above_point.circle_event is not None:
        arc_above_point.circle_event.fausse_alerte()

    # 3. Remplacer la feuille par le nouveau sous-arbre représentant les deux nouvelles intersections
    point_j = arc_above_point.origine
    breakpoint_gauche = Breakpoint(breakpoint=(point_j, point_i))
    breakpoint_droite = Breakpoint(breakpoint=(point_i, point_j))

    root = NoeudInterne(breakpoint_gauche)
    root.gauche = NoeudFeuille(Arc(origine=point_j, circle_event=None))
```

```python
if breakpoint_droite.s_intersectent():
    root.droite = NoeudInterne(breakpoint_droite)
    root.droite.gauche = NoeudFeuille(new_arc)
    root.droite.droite = NoeudFeuille(Arc(origine=point_j, circle_event=None))
else:
    root.droite = NoeudFeuille(new_arc)

self.arbre_front = arc_node_above_point.remplacer_feuille(replacement=root, racine=self.arbre_front)

A, B = point_j, point_i
AB = breakpoint_gauche
BA = breakpoint_droite
AB.edge = HalfEdge(B, origine=AB)
BA.edge = HalfEdge(A, origine=BA, jumelle=AB.edge)
self.edges.append(AB.edge)
B.arete_entree = B.arete_entree or AB.edge
A.arete_entree = A.arete_entree or BA.edge

# 5. On regarde si les breakpoints convergent avec les arcs à gauche et à droite
if not breakpoint_droite.s_intersectent():
    return

node_a, node_b, node_c = root.gauche.predecesseur, root.gauche, root.droite.gauche
node_c, node_d, node_e = node_c, root.droite.droite, root.droite.droite.successeur

self._verif_cercles((node_a, node_b, node_c), (node_c, node_d, node_e))

# 10. Rééquilibrer l'arbre
self.arbre_front = Arbre_AVL.equilibrer_et_propager(root)
```

```python
def traitement_circle_event(self, event: CircleEvent):

    # 1. Supprimer la feuille f qui reorésente l'arc disparaissant de T.
    arc = event.arc_pointer.data
    if arc in self._arcs:
        self._arcs.remove(arc)
    arc_node: NoeudFeuille = event.arc_pointer
    predecesseur = arc_node.predecesseur
    successeur = arc_node.successeur

    # Mise à jour des breakpoints
    self.arbre_front, updated, removed, gauche, droite = self._update_breakpoints(
        self.arbre_front, self.sweep_line, arc_node, predecesseur, successeur)

    if updated is None: return

    # Retirer tous les circle events impliquant l'arc de la file de priorité.
    def remove(neighbor_event):
        if neighbor_event is None:
            return None
        return neighbor_event.fausse_alerte()

    remove(predecesseur.get_value().circle_event)
    remove(successeur.get_value().circle_event)

    # 2. Créer les half-edges
    convergence_point = event.centre
    v = Vertex(convergence_point.xd, convergence_point.yd)
    self._vertices.add(v)
```

```python
updated.edge.origine = v
removed.edge.origine = v
v.connected_edges.append(updated.edge)
v.connected_edges.append(removed.edge)

# Récupérer les points incidents
breakpoint_a, breakpoint_b = updated.breakpoint

# Créer une nouvelle arête qui part de v et qui va vers le nouveau breakpoint
new_edge = HalfEdge(breakpoint_a, origine=v, jumelle=HalfEdge(breakpoint_b, origine=updated))
gauche.edge.jumelle.set_suiv(new_edge)
droite.edge.jumelle.set_suiv(gauche.edge)
new_edge.jumelle.set_suiv(droite.edge)
self.edges.append(new_edge)
v.connected_edges.append(new_edge)

# Le breakpoint mis à jour pointe vers la nouvelle arête
updated.edge = new_edge.jumelle

# 3. On regarde si le breakpoint converge avec les précédents arcs gauche et droite.
prec_gauche = predecesseur
prec_droite = successeur

node_a, node_b, node_c = prec_gauche.predecesseur, prec_gauche, prec_gauche.successeur
node_d, node_e, node_f = prec_droite.predecesseur, prec_droite, prec_droite.successeur

self._verif_cercles((node_a, node_b, node_c), (node_d, node_e, node_f))
```

```python
def _verif_cercles(self, triple_gauche, triple_droite):
    node_a, node_b, node_c = triple_gauche
    node_d, node_e, node_f = triple_droite

    gauche_event = CircleEvent.creer_circle_event(node_a, node_b, node_c, sweep_line=self.sweep_line)
    droite_event = CircleEvent.creer_circle_event(node_d, node_e, node_f, sweep_line=self.sweep_line)

    # On regarde si les cercles convergent
    if gauche_event:
        if not Geometrie.angle_est_horaire(node_a.data.origine, node_b.data.origine, node_c.data.origine,
                                            gauche_event.centre):
            gauche_event = None

    if droite_event:
        if not Geometrie.angle_est_horaire(node_d.data.origine, node_e.data.origine, node_f.data.origine,
                                            droite_event.centre):
            droite_event = None

    if gauche_event is not None:
        self.event_queue.put(gauche_event)
        node_b.data.circle_event = gauche_event

    if droite_event is not None and gauche_event != droite_event:
        self.event_queue.put(droite_event)
        node_e.data.circle_event = droite_event

    return gauche_event, droite_event
```

```python
@staticmethod
def calculer_angle(point, center) -> float:
    dx = point.xd - center.xd
    dy = point.yd - center.yd
    return np.math.degrees(np.math.atan2(dy, dx)) % 360

@staticmethod
def angle_est_horaire(a, b, c, center) -> bool:
    angle_1 = Geometrie.calculer_angle(a, center)
    angle_2 = Geometrie.calculer_angle(b, center)
    angle_3 = Geometrie.calculer_angle(c, center)

    est_direct = (
        angle_3 - angle_1) % 360 > (angle_3 - angle_2) % 360

    if est_direct:
        return False

    return True
```

```python
def test_naif(nom_carte: str, precision: float, bounding_size, debut, arrivee) -> dict:

    Settings.PRECISION = precision
    temps = {}

    time_start = perf_counter_ns()
    carte = CarteNaif.from_scratch(nom_carte, bounding_size)
    time_end = perf_counter_ns()
    print(f"Construction naif:{time_end - time_start} ns")
    temps[CONSTRUCTION_NAIF] = time_end - time_start

    time_start = perf_counter_ns()
    try:
        _ = carte.dijkstra(debut, arrivee)
    except CheminNonTrouve as err:
        print(err)
    time_end = perf_counter_ns()
    temps[CHEMIN_DIJKSTRA] = time_end - time_start

    time_start = perf_counter_ns()
    try:
        _ = carte.a_star(debut, arrivee)
    except CheminNonTrouve as err:
        print(err)
    time_end = perf_counter_ns()
    temps[CHEMIN_ASTAR] = time_end - time_start

    return temps
```