



Déplacement d'un mobile ponctuel sur un terrain

Léo SAMUEL



Sommaire

I – Théorie

- Description ensembliste
- Théorie des graphes

II – Implémentation

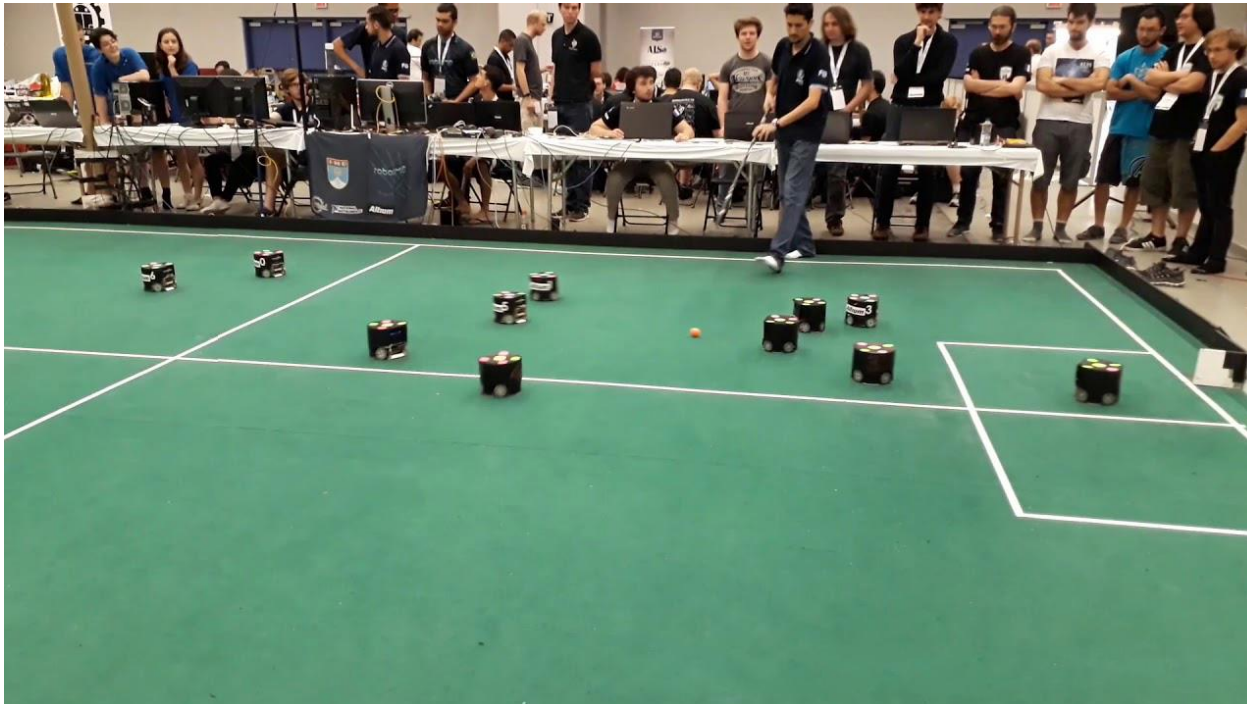
- Comparaisons
- Influence de l'évaluation heuristique
- Complexité

III – Expérimentation

- Dispositif
- Résultat

Introduction

Comment permettre la navigation d'un robot sur un terrain de football ?

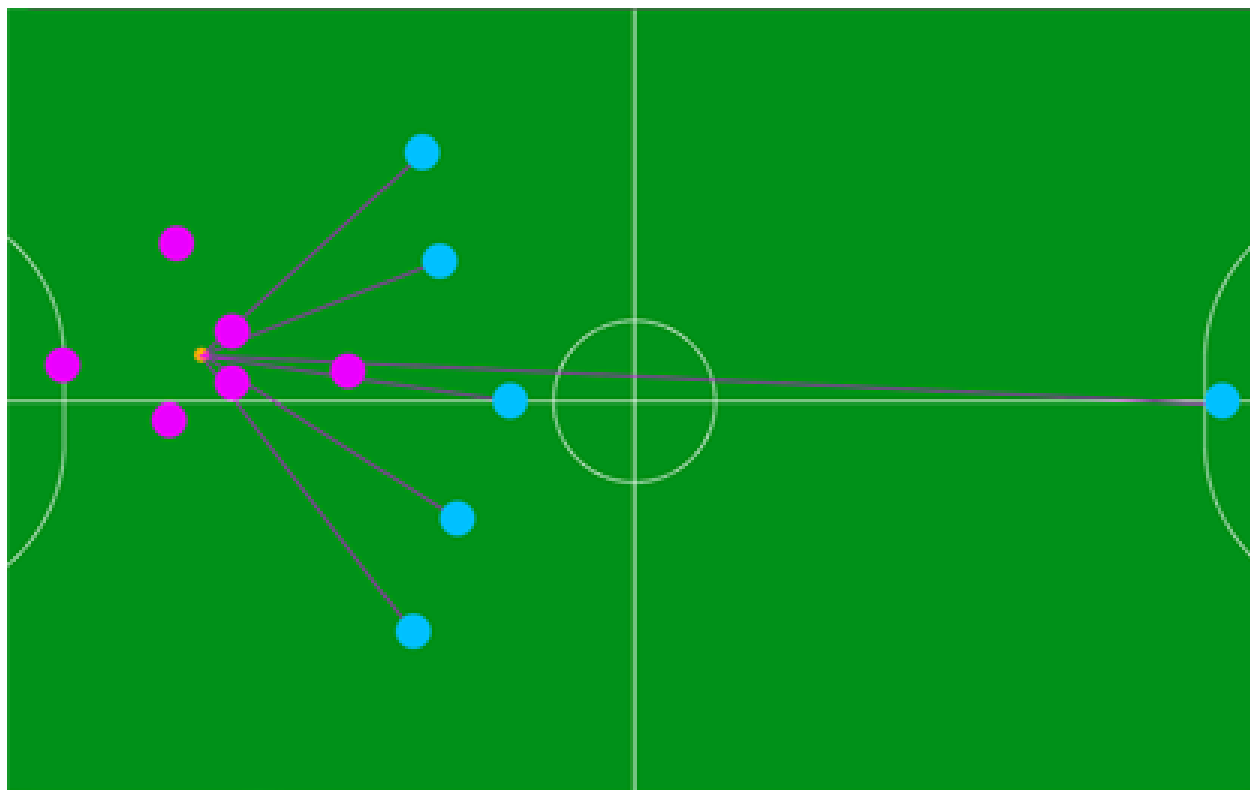


Source : robocup.org

Source : Chaine Youtube « Rhoban » : https://youtu.be/oKCEoa_XXFo?t=50

I – Théorie

Description ensembliste





I – Théorie

Description ensembliste

Ensemble convexe :

$$\forall X, Y \in A, \forall t \in [0,1], tx + (1 - t)y \in A$$

Ensemble connexe par arcs :

$$\forall X, Y \in A, \exists \varphi \in \mathcal{C}([0,1], A), \forall t \in [0,1], \begin{cases} \varphi(0) = X \\ \varphi(1) = Y \\ \forall t \in [0,1], \varphi(t) \in A \end{cases}$$



I – Théorie

Théorie des graphes

Définition 1 : *Évaluation Heuristique*

On appelle évaluation heuristique une application $\mathcal{H}: \mathcal{S} \rightarrow \mathbb{R}$

Définition 2 : *Nœud*

On note \mathcal{C} les coordonnées du nœud

On note \mathcal{H} l'évaluation heuristique du nœud

On appelle nœud le couple $(\mathcal{C}, \mathcal{H})$



I – Théorie

Théorie des graphes

Définition 3 :

On dit que deux nœuds N_1 et N_2 sont égaux lorsqu'ils ont les mêmes coordonnées.

On note alors $N_1 = N_2$.

On dit qu'un nœud N_1 est inférieur à un nœud N_2 lorsque l'évaluation heuristique de N_1 est inférieure à celle de N_2 .

On note $N_1 < N_2$.



II – Implémentation

Première implémentation : Tableaux

Inconvénient : Nécessite de trier la liste

Avantage : Simple à implémenter avec une structure connue



II – Implémentation

Deuxième implémentation : Tableau et tas

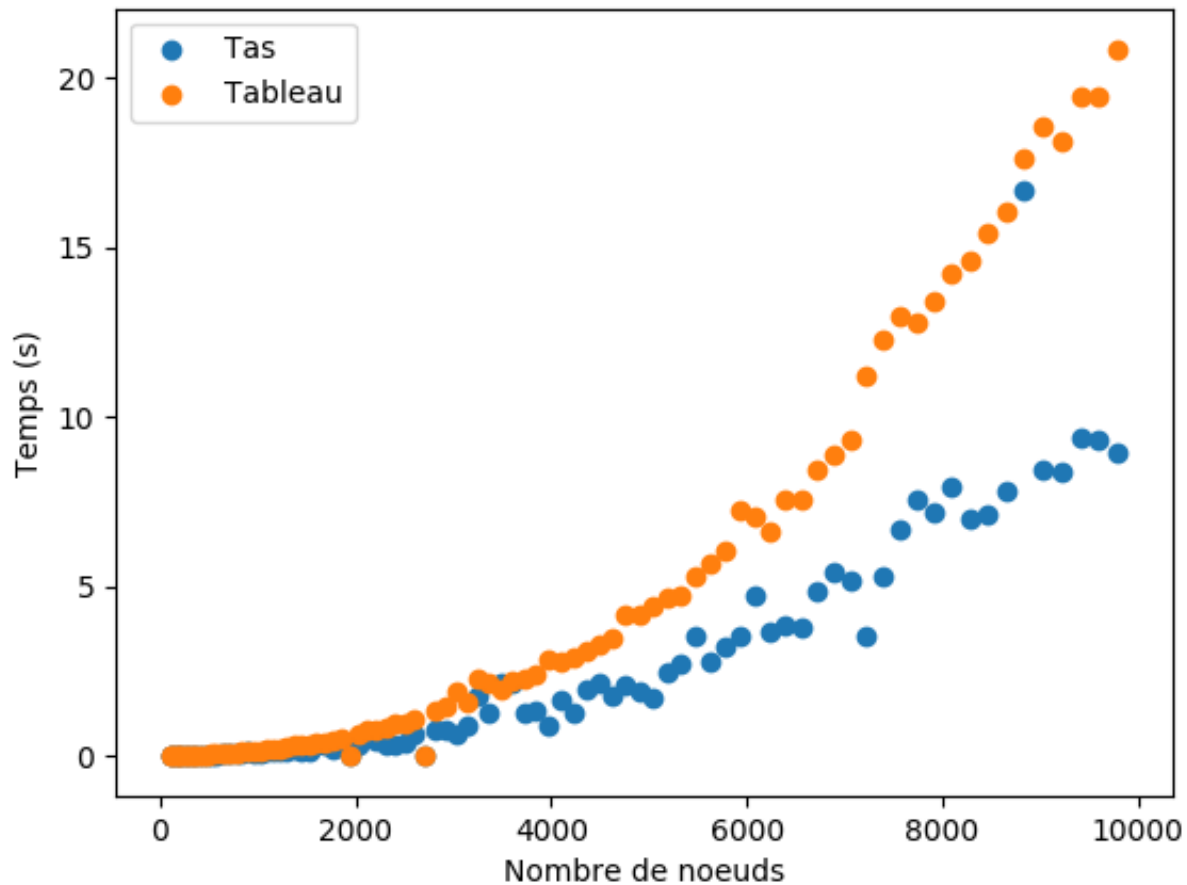
Inconvénient : Ne change pas la complexité de la fonction

Avantage : Plus efficace



II – Implémentation

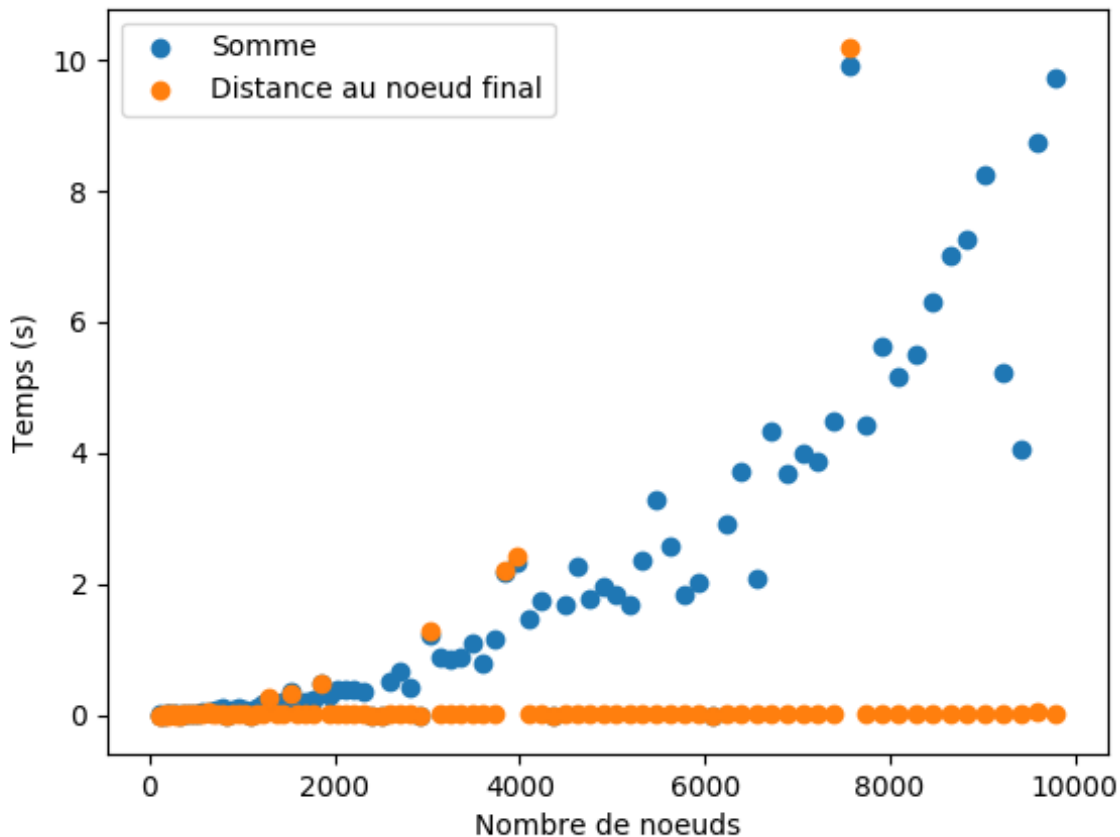
Comparaison





II – Implémentation

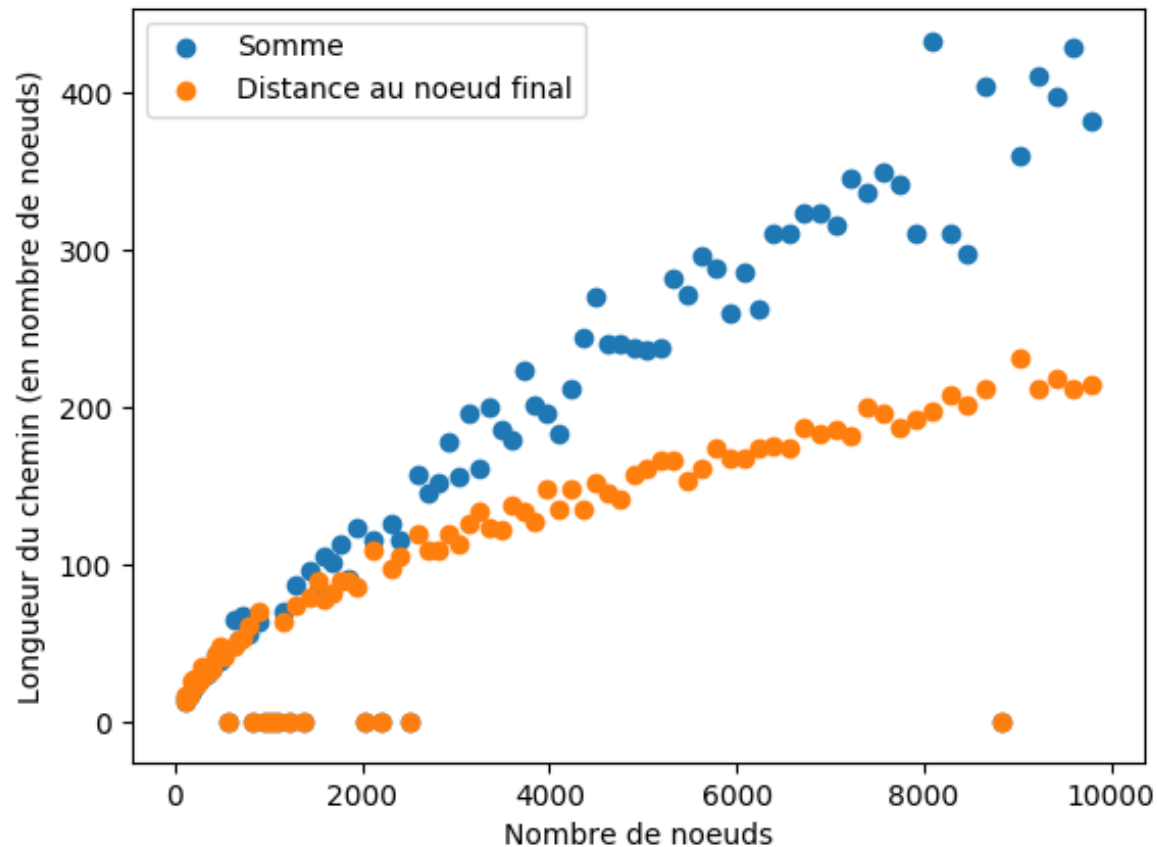
Influence de l'évaluation heuristique





II – Implémentation

Influence de l'évaluation heuristique





II – Implémentation

Complexité

Définition 4 :

On appelle facteur d'embranchement le nombre de fils d'un nœud

On appelle profondeur d'un nœud la longueur du chemin entre ce dernier et le nœud de départ



II – Implémentation

Complexité

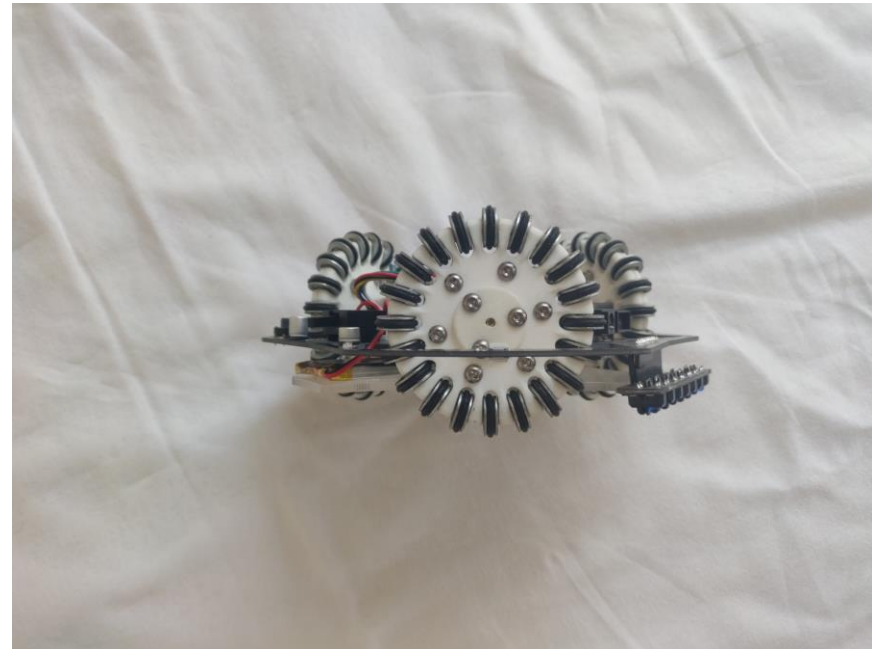
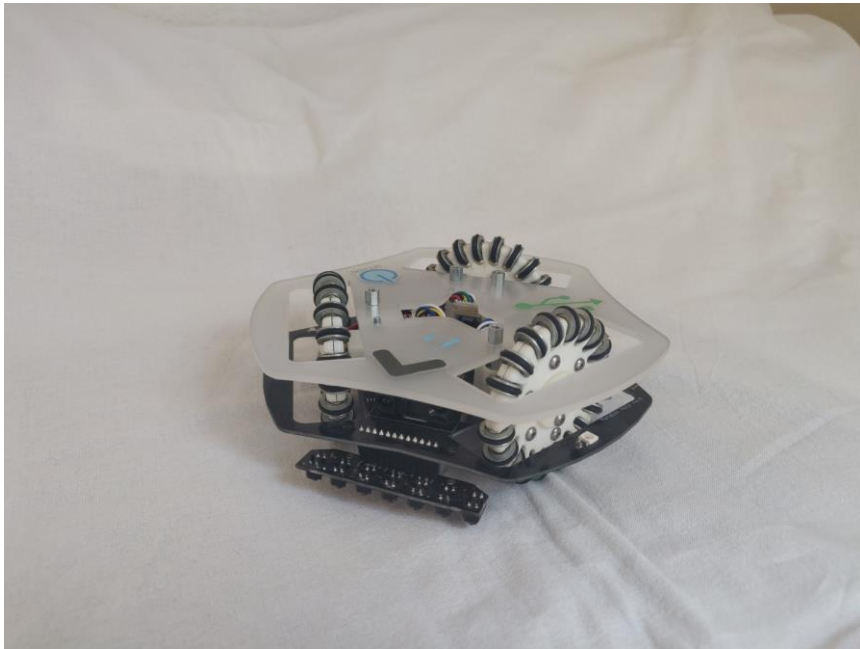
Proposition 1 :

Soit $n \in \mathbb{N}$. Le nombre de nœuds de profondeur n dans un arbre de facteur d'embranchement b est b^n

La complexité est alors : $O(b^n)$



III – Expérimentation



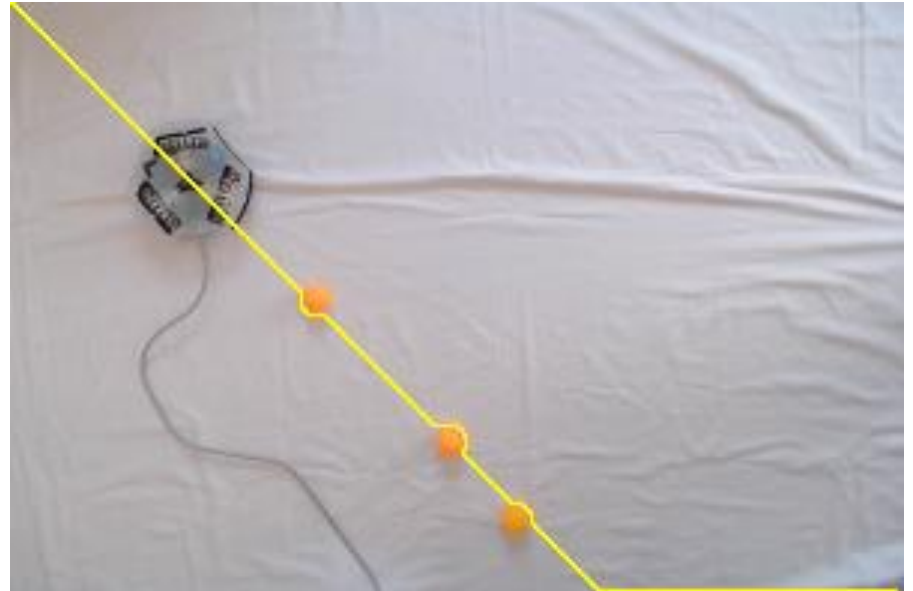
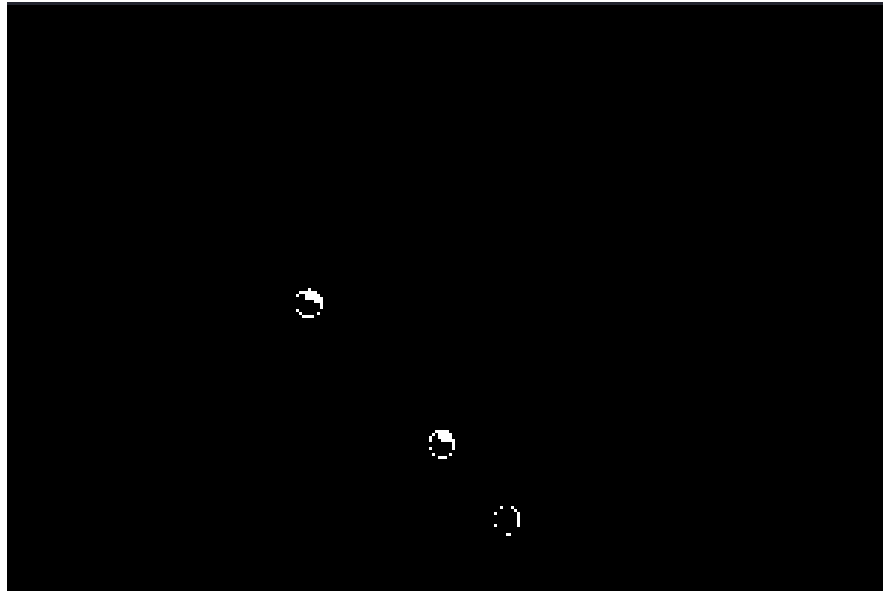


III – Expérimentation





III – Expérimentation





Conclusion

Chemin discret


Déplacement saccadé

Modélisation de la trajectoire pour plus de fluidité




Annexes


```
class Noeud:
    def __init__(self, position:(), parent:()):
        self.position = position
        self.parent = parent
        self.g = 0 # Distance au Noeud de départ
        self.h = 0 # Distance au Noeud de fin
        self.f = 0 # Cout total
    def __eq__(self, other):
        return self.position == other.position
    def __lt__(self, other):
        return self.f < other.f
```



```
def add_to_à_Visiter(à_visiter, voisin):  
    for noeud in à_visiter:  
        if (voisin == noeud and voisin.f >= noeud.f):  
            return False  
    return True  
  
def Distance(a,b):  
    x = b.position[0]-a.position[0]  
    y = b.position[1]-a.position[1]  
    return ((x)**2 + (y)**2)**(1/2)
```




```
def carteFromMatrix(M):  
    largeur = len(M[0])  
    hauteur = len(M)  
    map = {}  
    deb,fin = (0,0),(0,0)  
  
    for j in range (hauteur):  
        for i in range (largeur):  
            map[(i, j)] = M[j][i]  
            if M[j][i]=='@':  
                deb = (i,j)  
            if M[j][i]=='$':  
                fin = (i,j)  
    return map,deb,fin,largeur,hauteur
```



```

def A_étoile(map, deb, fin):
    à_visiter = []
    déjà_vu = []
    start_node = Noeud(deb, None)
    goal_node = Noeud(fin, None)
    à_visiter.append(start_node)
    while len(à_visiter) > 0:
        à_visiter.sort()
        current_node = à_visiter.pop(0)
        déjà_vu.append(current_node)
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path
        (x, y) = current_node.position
        Voisins = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for next in Voisins:
            map_value = map.get(next)
            if(map_value == '#'):
                continue
            voisin = Noeud(next, current_node)
            if(voisin in déjà_vu):
                continue
            voisin.g = Distance(voisin, start_node)
            voisin.h = Distance(voisin, goal_node)
            voisin.f = voisin.g + voisin.h
            if(add_to_à_Visiter(à_visiter, voisin) == True):
                à_visiter.append(voisin)
    return None

```



```

def A_étoile(map, deb, fin):
    à_visiter = []
    déjà_vu = []
    start_node = Noeud(deb, None)
    goal_node = Noeud(fin, None)
    heapq.heappush(à_visiter, start_node)
    while len(à_visiter) > 0:
        current_node = heapq.heappop(à_visiter)
        déjà_vu.append(current_node)
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path
        (x, y) = current_node.position
        Voisins = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for next in Voisins:
            map_value = map.get(next)
            if (map_value == '#'):
                continue
            voisin = Noeud(next, current_node)
            if (voisin in déjà_vu):
                continue
            voisin.g = Distance(voisin, goal_node)
            voisin.h = Distance(voisin, start_node)
            voisin.f = voisin.g + voisin.h
            if (add_to_à_Visiter(à_visiter, voisin) == True):
                heapq.heappush(à_visiter, voisin)
    return None


```



```


def analyse(frame):
    x,y=-1,-1
    flag = False
    frame = imutils.resize(frame, width=tailleImage)
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, orangeLower, orangeUpper)
    cnts = cv2.findContours(mask.copy(),
        ↳ cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]
    center = None
    if len(cnts) > 0:
        c = max(cnts, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(c)
        if radius > 10:
            cv2.circle(frame, (int(x), int(y)),
                ↳ int(radius), (0, 255, 255), 2)
            flag = True
        else:
            radius = -1
            x = -1
            y = -1
            flag = False
    else:
        radius = -1
        x = -1
        y = -1
        flag = False
    return flag, (x,y), frame, mask

```

```
holo = Holobot(sys.argv[1], 115200)
camera = cv2.VideoCapture(0)

def deplacement_élémentaire(pos, dest ,poids=30 ,tps=0.1):
    x=dest[0]-pos[0]
    y=dest[1]-pos[1]
    holo.control(poids*x,poids*x,0)
    time.sleep(tps)
```



```
def deplacement():
    (grabbed, frame) = camera.read()
    B, coord, frame, mask = analyse(frame)
    mat = nvmat(len(mask), len(mask[0]))
    map, deb, fin, largeur, hauteur = carteFromMatrix(mat)
    path = A_etoile(map, deb, fin)
    dest = path[1]
    x = coord[0]
    y = coord[1]
    if x >= 0 and y >= 0:
        deplacement(coord, dest)
```