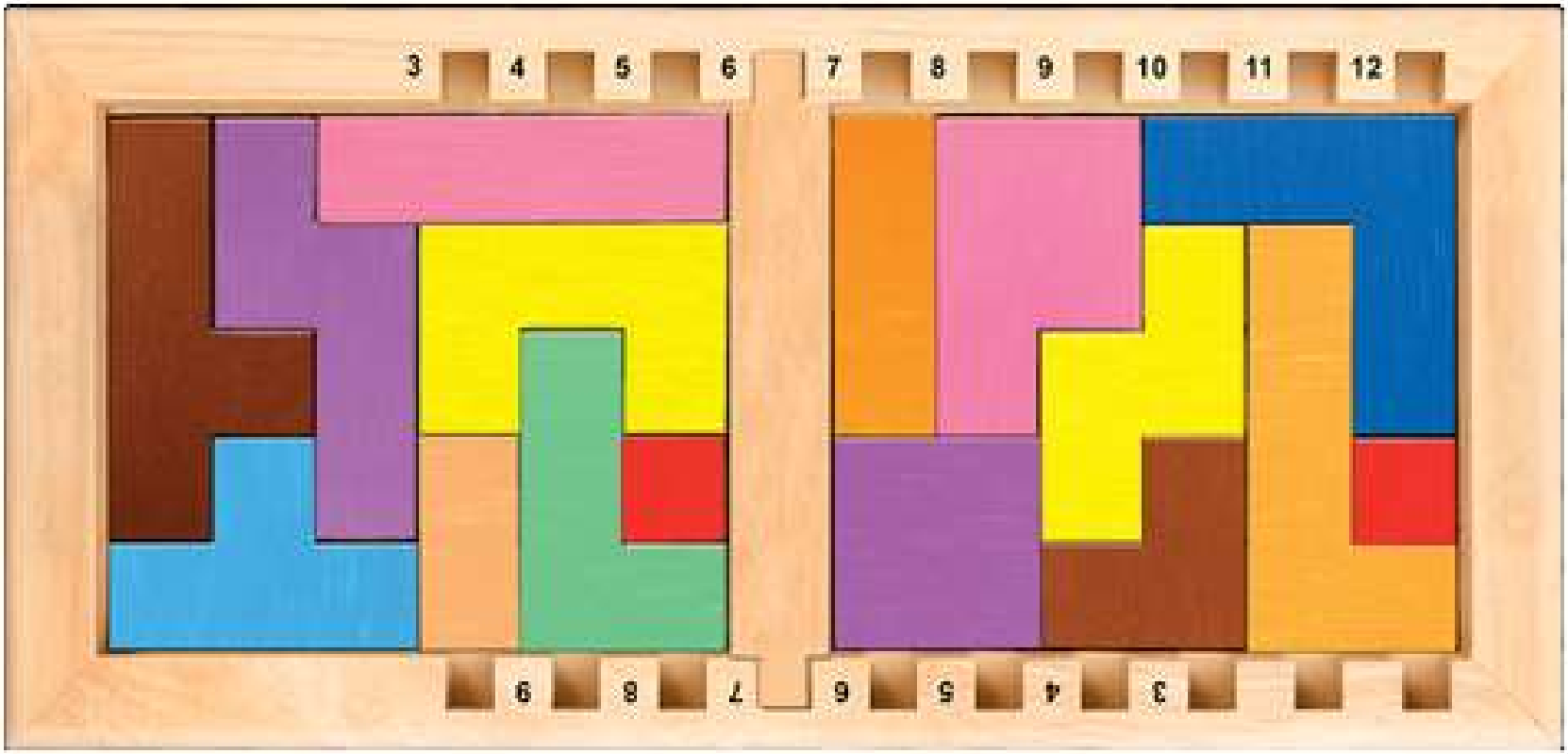
KATAMINO

Recherche de solutions dans une grille finie par simulation numérique

SOMMAIRE

- Introduction
- Réalisation d’ un solveur de Katamino
  - Par Force Brute
  - Mise en place d’un comptage de la complexité
  - Optimisation gloutonne de Force Brute
  - Par Backtracking

INTRODUCTION :



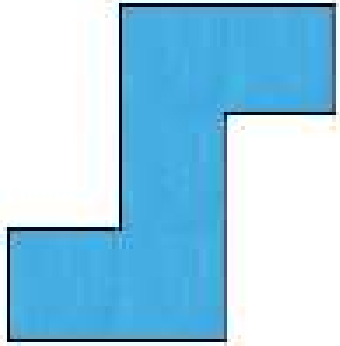


Figure 1

INTRODUCTION : Dénomination

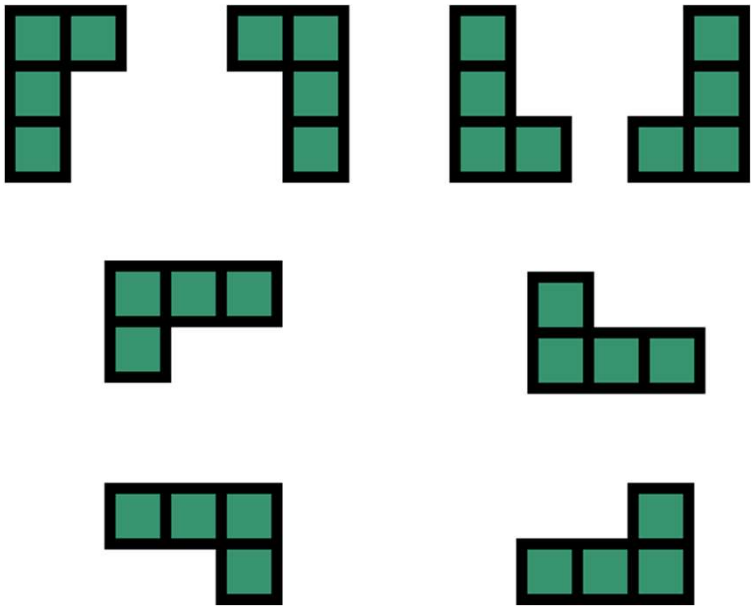


Figure 2

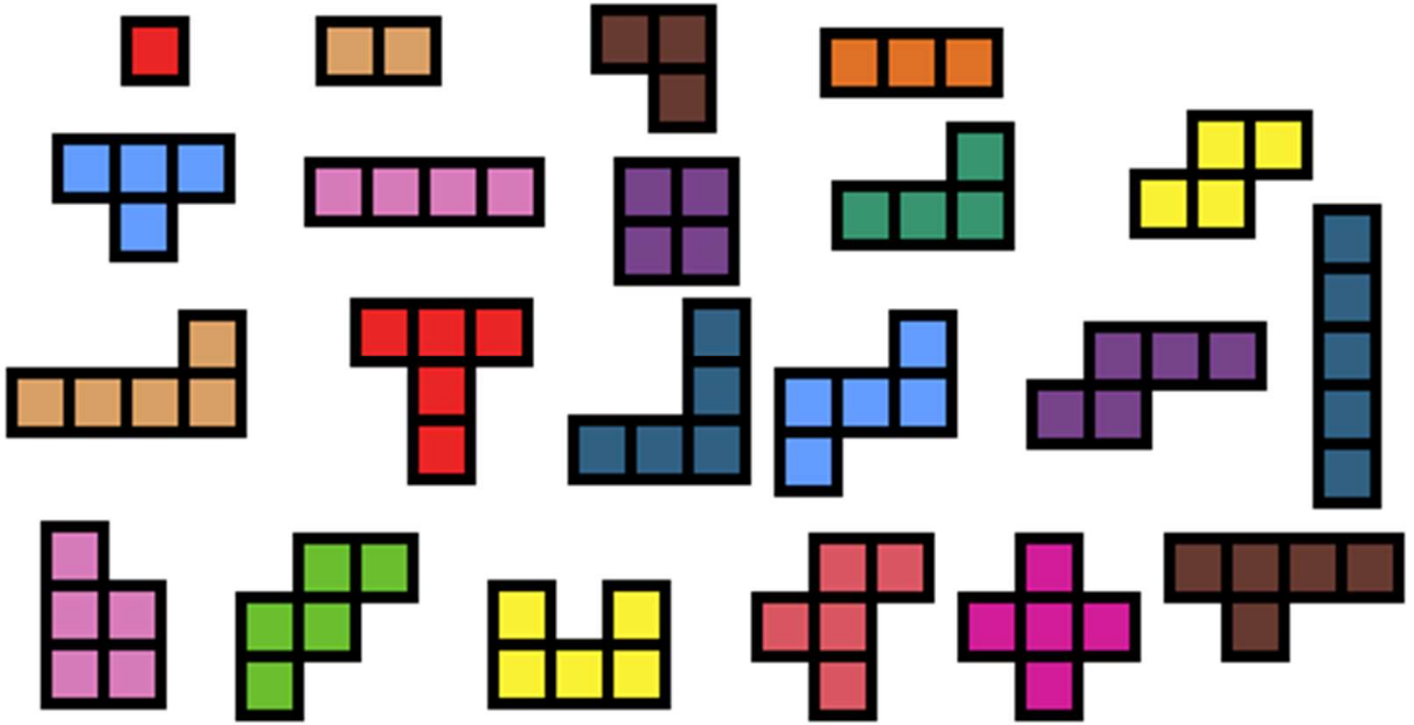


Figure 3

INTRODUCTION : Histoire

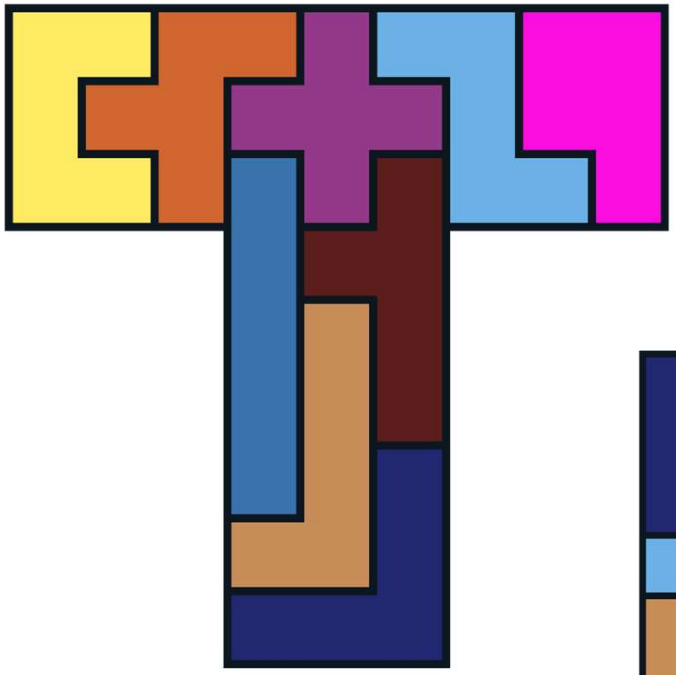


Figure 4

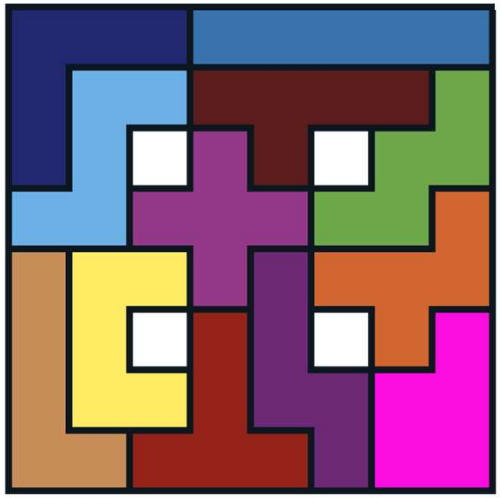
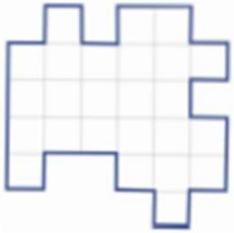


Figure 5

4. Collage (coefficient 4)

Lina découpe cinq figures identiques qui ont la forme représentée à gauche. Elle les colle de manière à recouvrir entièrement cette grande figure.



Retrouvez son collage. Attention : On peut tourner les pièces à plat mais pas les retourner.

Figure 6: Un problème de la finale FFJM mai 2023

INTRODUCTION : Représentation des kataminos

```
[[ 7 , 7 , 7 , 4 , 4 , 4 , 4 , 1 , 1 ],
 [ 5 , 7 , 7 , 2 , 2 , 2 , 4 , 1 , 9 ],
 [ 5 , 15, 15, 2 , 6 , 6 , 1 , 1 , 9 ],
 [ 5 , 5 , 10, 2 , 6 , 3 , 3 , 9 , 9 ],
 [ 5 , 10, 10, 10, 6 , 6 , 3 , 3 , 3 ]]
```

Figure 8

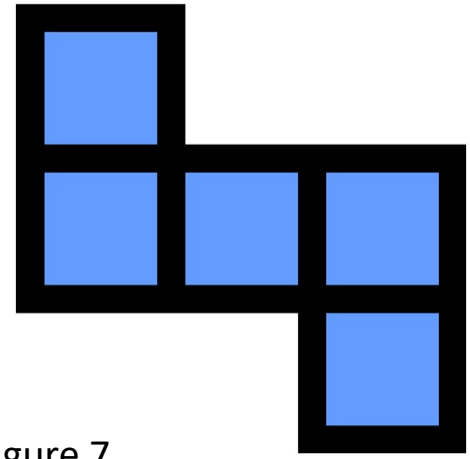


Figure 7

```
Z5=[[1,0,0],
    [1,1,1],
    [0,0,1]]
```

SOLVEUR DE KATAMINO : Force Brute

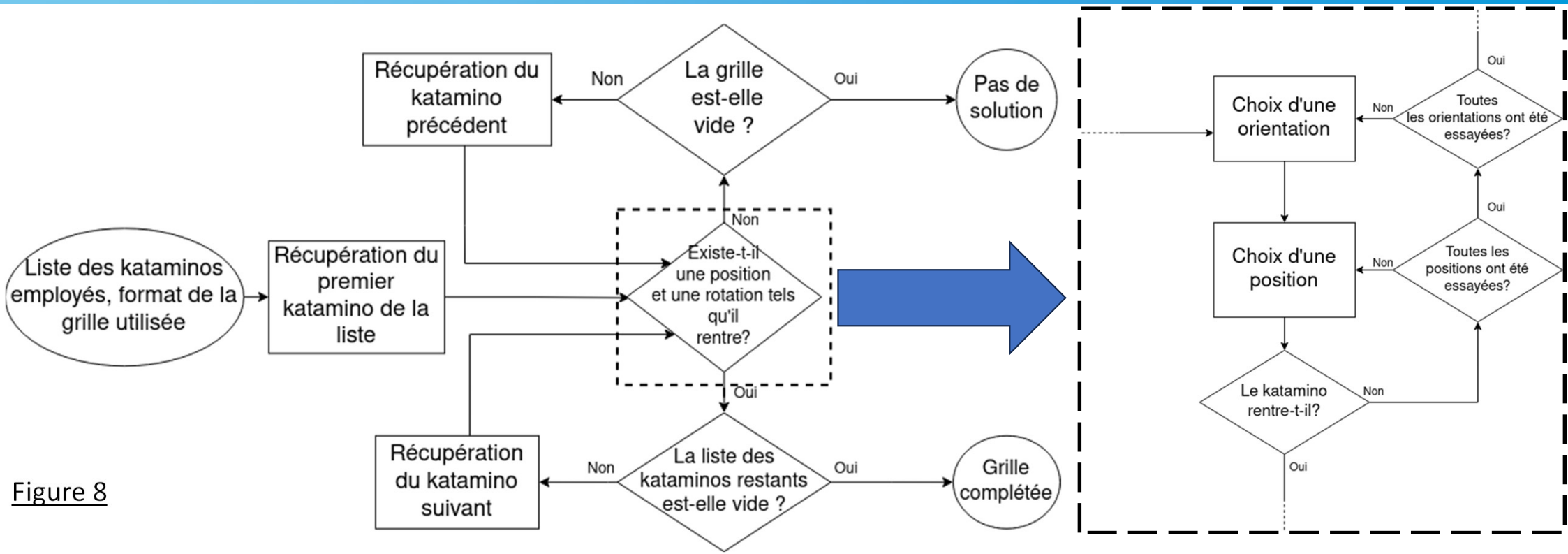
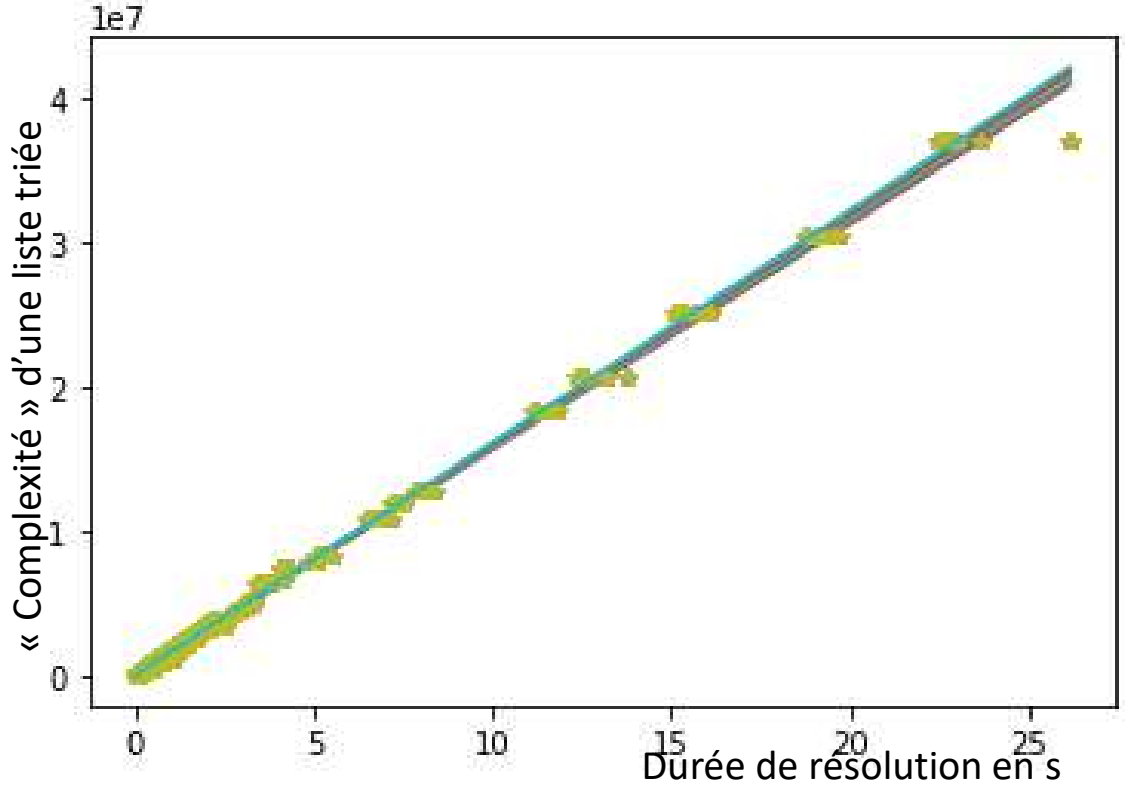


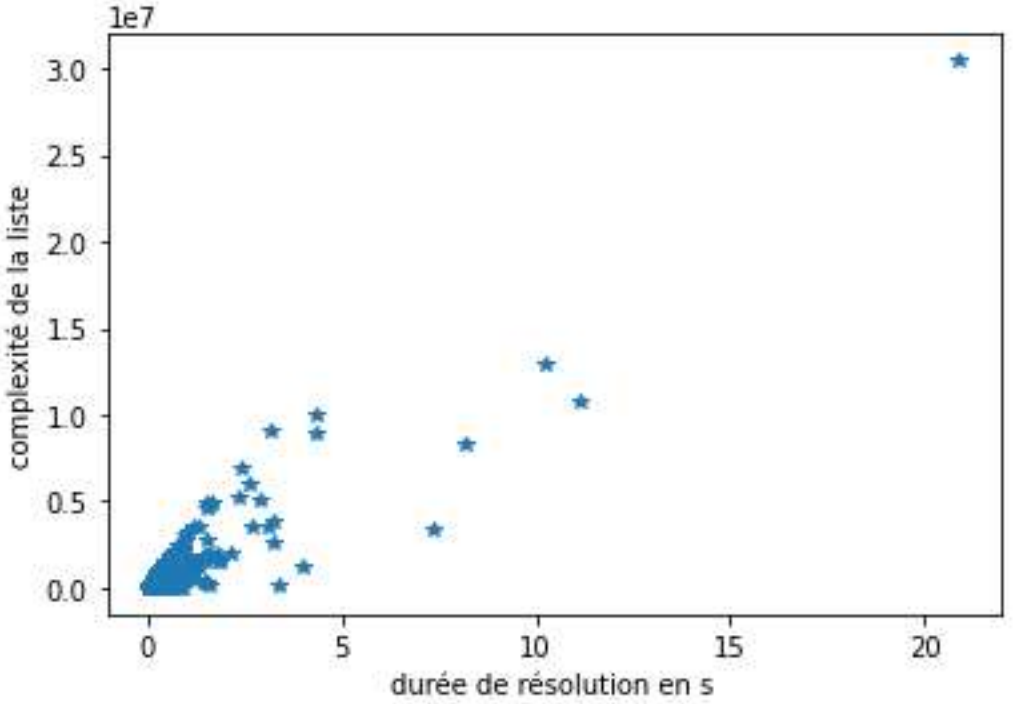
Figure 8

Complexité : $O(N!\,A^2)$

Mise en place d'un comptage de la complexité



durée de résolution en fonction de la « Complexité » pour 400 listes rentrant dans une grille d'aire 20 répété 10 fois

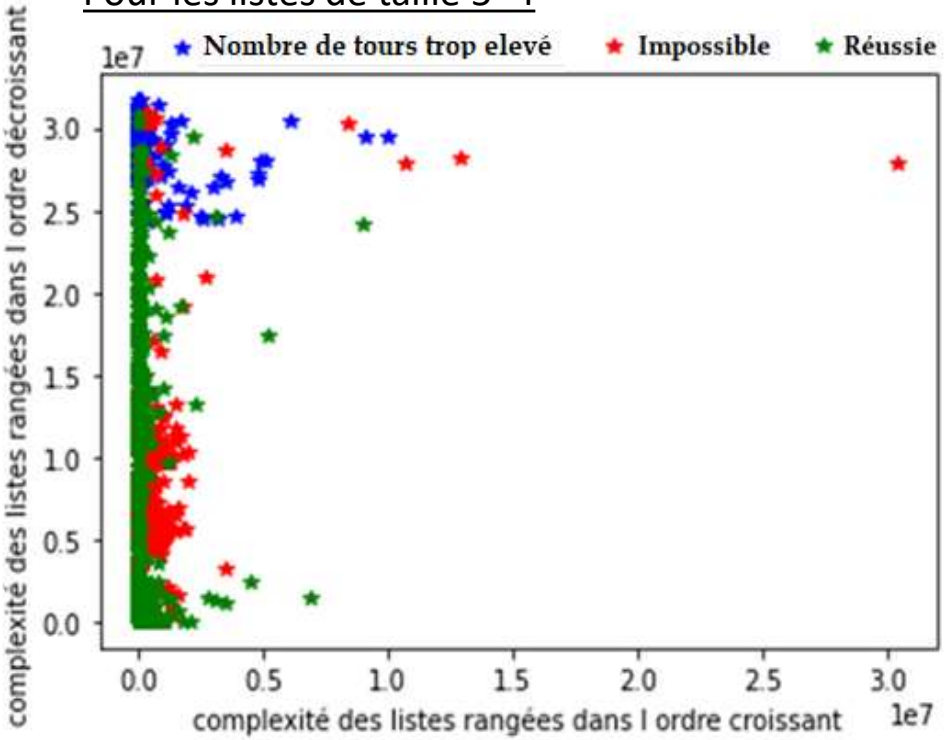


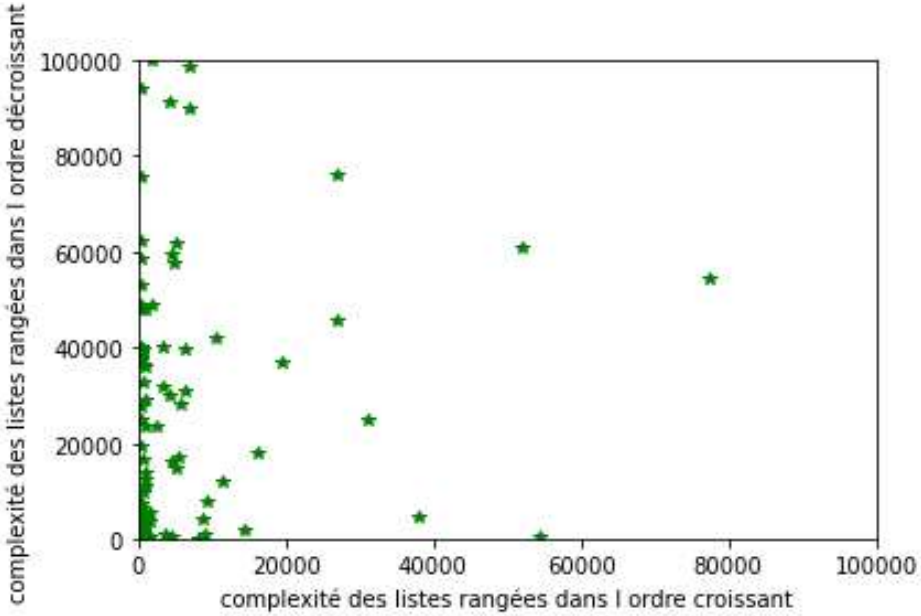
durée de résolution en fonction de la « Complexité » pour 2000 listes rentrant dans une grille d'aire 20 répété 100 fois

Optimisation gloutonne de Force Brute

On donne des valeurs aux kataminos et on trie la liste

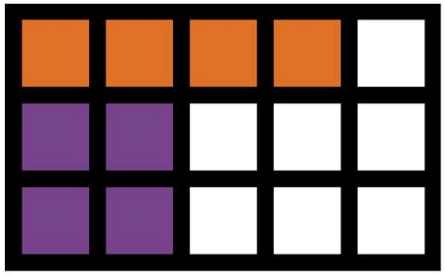
Pour les listes de taille 5*4

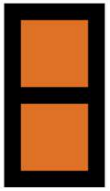




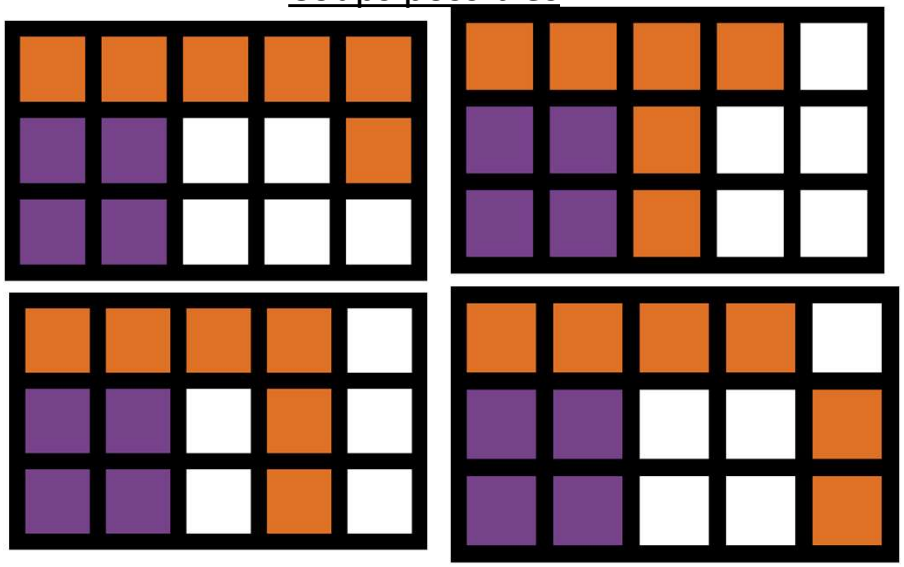
SOLVEUR DE KATAMINO : Par retour sur trace

Position de départ





Coups possibles



Complexité de la résolution par retour sur trace

Formule 1

$$C(N) = A + A^N(N-1) + C(N-1)A$$

Formule 2

$$C(N) = A^N + \frac{A-N}{A-1}A^N + \frac{N(N-1)}{2}A^N + C(0)$$

Formule 3

$$C(N) = O(N^2 A^N)$$

Ouverture

- Trouver une méthode plus optimisée
- Fournir une méthodologie de résolution gagnante pour humain (statistiques)
- Etendre la résolution à tout type de polyominos et plateaux

ANNEXE 1/15

```
#ROTATION D'UNE PIECE DE 90°
def tourne90(Piece):
    "tourne une pièce de 90° dans le sens horaire"
    "Piece est une matrice"
    NPiece=[ [0 for _ in range (len(Piece))] for _ in range (len(Piece[0]))]
#on #crée une matrice vide
    for i in range (len(Piece)): #lignes
        for j in range (len(Piece[0])): #colonnes
            NPiece[j][i]=Piece[len(Piece)-1-i][j] #on prend la transposée et la
#symétrie par rapport à l'axe des ordonnées
    Piece=NPiece
    return Piece
```

```
#SYMETRIE par l'ordonnée
def retourne(Piece):
    "effectue la symétrie une pièce par l'ordonnée en place"
    "Piece est une matrice"
    NPiece=[ [0 for i in range (len(Piece[0]))] for _ in range
(len(Piece))]
    for i in range (len(Piece)):
        for j in range (len(Piece[0])):
            NPiece[i][j]=Piece[i][len(Piece[0])-j-1]
    return NPiece
```

ANNEXE 2/15

```
#AJOUT D'UNE PIECE AU PLATEAU
def ajout(Piece, Plateau=PB, l=0, c=0):
    "ajoute une pièce à un plateau"
    "Piece et Plateau sont des matrices, Plateau est par défaut une matrice nulle, l et c int par défaut 0"
    # (l,c) correspond aux coordonnées de la case (0,0) de la piece dans le plateau
    Nplateau=np.copy(Plateau) #on fait un nouveau plateau
    for i in range (len(Piece)): #lignes
        for j in range (len(Piece[0])): #colonnes
            if Piece[i][j]!=0:
                if Nplateau[i+l][j+c]==0: #si la case du plateau est vide
                    Nplateau[i+l][j+c]=Piece[i][j] #on pose la case de piece
                else:
                    return False #sinon la pièce ne peut pas être posée
    return Nplateau #on retourne le nouveau plateau
```

```
#APPLIQUE UNE LISTE DE PIECES A UN PLATEAU
def applique(Liste, Plateau):
    "applique la liste de pièces à un plateau par défaut une matrice nulle"
    "Liste contient une liste de [piece,tourne,symetrie,ligne,colonne]"
    for i in range (len(Liste)):
        Plateau = ajout(Liste[i][0], Plateau, Liste[i][3], Liste[i][4])
    return Plateau
```

ANNEXE 3/15 force Brute

```
Résolution par force brute

def force_brute(Liste, Plateau):
    "résoud un problème avec les pièces données dans la liste"
    "teste toutes les possibilitées avant d'en trouver une qui fonctionne"
    L=np.copy(Liste) #liste des pieces de départ
    Lparams=[] #contient [piece,tourne,symetrie,ligne,colonne]
    s,t,i,j=0,0,0,0 #compteurs de symetrie, tourne, ligne et colonne
    k = 0 #compteur de tours
    while len(L)!=0:
        k += 1
        posee=False #au départ la piece n'est pas posée
```

ANNEXE 4/15 force Brute

```
#on teste toutes les possibilités de positionnement de la piece
while s<2 and not posee: #symetrie
   while t<4 and not posee: #tourne
      while i<=len(Plateau)-len(L[0]) and not posee: #ligne
         while j<=len(Plateau[0])-len(L[0][0]) and not posee: #colonne
            nP = ajout(L[0],Plateau,i,j)
            if type(nP)!=bool :
               Plateau = nP
               posee=True
               Lparams.append([L[0],s,t,i,j])  #[piece,tourne,symetrie,ligne, colonne]
            j+=1
         j=0
         i+=1
      i=0
      L[0]=tourne90(L[0])
      t+=1
   t=0
   L[0]=retourne(L[0])
   s+=1
s=0
```

ANNEXE 5/15 force Brute

```
 if not posee: #si la piece n'est pas posée
        if len(Lparams)==0:
            return Plateau, "échec"

        L=Liste[len(Liste)-len(L)-1:] #on ajoute la pièce d'avant aux pieces non posées
        L[0], s, t, i, j =Lparams[-1][0], Lparams[-1][1], Lparams[-1][2], Lparams[-1][3], Lparams[-1][4] #pour
reprendre la où on s'était arreté avec la pièce davant
        Lparams=Lparams[:-1] #on retire la pièce d'avant des pièces posées
```

ANNEXE 6/15
Force brute

```
if i>len(Plateau)-len(L[0]):
            i=0
            if j>len(Plateau[0])-len(L[0][0]):
               j=0
               if t>3:
                  t=0
                  if s>0:
                     #Toutes les positions ont été testées ...
                     ()
                  else:
                     s+=1
                     L[0]=retourne(L[0])
               else:
                  t+=1
                  L[0]=tourne90(L[0])
            else:
               j+=1
         else:
            i+=1
         Plateau=applique(Lparams,Plateau) #on reprend le plateau sans la #pièce d'avant
      else:
         L=L[1:] # liste des pieces restantes

   return Plateau
```

ANNEXE 7/15
Force brute

```
if i>len(Plateau)-len(L[0]):
            i=0
            if j>len(Plateau[0])-len(L[0][0]):
               j=0
               if t>3:
                  t=0
                  if s>0:
                     #Toutes les positions ont été testées ...
                     ()
                  else:
                     s+=1
                     L[0]=retourne(L[0])
               else:
                  t+=1
                  L[0]=tourne90(L[0])
            else:
               j+=1
         else:
            i+=1
         Plateau=applique(Lparams,Plateau) #on reprend le plateau sans la #pièce d'avant
      else:
         L=L[1:] # liste des pieces restantes

   return Plateau
```

ANNEXE 8/15 Validation du modèle

```
from os import chdir
import time

Ck=[]
Ct=[]
Liste20=open('combi - Copie/test20.txt','r')
Listes20=Liste20.readlines()
Liste20.close()
for i in range (len(Listes20)):
    Listes20[i]=eval(Listes20[i].strip())

for i in range(len(Listes20)):
    tri_insertion(Listes20[i])
    compl=0
    force_brute(Listes20[i])
    Ck.append(compl)

Ct=[[] for _ in range(len(Listes20))]
```

```
for k in range (100):
t1=time.time()
        force_brute(Listes20[i])
        t2=time.time()
        Ct[i].append(t2-t1)

        t1=time.time()
        force_brute(Listes20[i],1)
        t2=time.time()
        Ct[i].append(t2-t1)

Ct2=[]
for i in range (len(Ct)):
    Ct2.append(np.mean(Ct[i]))

plt.plot(Ct2,Ck,'*')
```

ANNEXE 9/15 Étude pratique de complexité

```
from os import chdir
def tri_insertion (L):
    n=len(L)
    for i in range (1,n):
        j=i
        x=L[i]

        while 0<j and x<L[j-1]:
            L[j]=L[j-1]
            j=j-1
        L[j]=x

def lenvers(L):
    n=len(L)
    for i in range (n//2):
        L[i],L[n-i-1]=L[n-i-1],L[i]
```

```
Liste20=open('combi - Copie/test20.txt','r')

Listes=Liste20.readlines()

Liste20.close()

for i in range (len(Listes)):
    Listes[i]=eval(Listes[i].strip())

#plateau
LONG=4
LARGE=5
PB=[[0 for _ in range(LARGE)]for _ in range (LONG)]


Cs2=[]  #listes des complexités des grilles solubles dans l'ordre croissant
Cs1=[]  #dans l'ordre décroissant
Cimp2=[] #listes des complexités des grilles n'admettant pas de solutions
Cimp1=[]
Coob1=[] #listes des complexités des grilles prenant trop de temps à résoudre
Coob2=[]
```

ANNEXE 10/15 Étude pratique de complexité

```
for i in range (1915):
    compl=0
    tri_insertion(Listes[i])
    P1=force_brute(Listes[i])
    compl1=compl    #complexité de la liste dans l'ordre croissant

    compl=0
    lenvers(Listes[i])
    P2=force_brute(Listes[i])
    compl2=compl

    if type(P1)==str or type(P2)==str: #si la grille n'admet pas de solution ou #que la résolution prend trop de temps
        if (type(P1)==str and P1=='impossible') or (type(P2)==str and P2=='impossible'): #si la grille n'admet pas de solutions
            Cimp1.append(compl1)    #on empile les complexités dans les listes appropriées
            Cimp2.append(compl2)
        else:
            Coob1.append(compl1)    #on empile les complexités dans les listes de
                                    #complexité des grilles prenant trop de temps à résoudre
            Coob2.append(compl2)
    else:
        Cs2.append(compl2) #la grille est résolue et on empile les complexités #dans les listes appropriées
        Cs1.append(compl1)
```

ANNEXE 11/15 Étude pratique de complexité

```
plt.plot(Cs1,Cs2,'g*')
plt.plot(Cimp1,Cimp2,'r*')
plt.plot(Coob1,Coob2,'b*')
plt.xlabel('complexité des listes rangées dans l'ordre croissant')
plt.ylabel('complexité des listes rangées dans l'ordre décroissant')
plt.show()
```

ANNEXE 12/15 Définition du type Katamino

```
class Katamino:
  def __init__(self, shape, p, sym, max_rot, x_coord=0, y_coord=0):
     self.s=shape   #matrice représentant le katamino
     self.x=x_coord #stocke la coordonnée x du point (0,0) du katamino dans le tableau
     self.y=y_coord #coordonnée y
     self.priority=p
     self.symmetry=sym #booléen vrai si le katamino vérifie une symétrie axiale
     self.max_r= max_rot #nombre de rotations par lesquels le katamino n'est pas invariant
     self.area=a    #aire du katamino
```

ANNEXE 13/15 Résolution par backtracking

```
def Coups_Possibles(Kata,Plateau,n):
    """renvoie l'ensemble des coups possibles à partir d'une position donnée, Plateau, avec une pièce Kata
    n est l'aire libre du plateau"""
    i=0
    K=deepcopy(Kata)    #on copie le katamino
    K.x,K.y,r=0,0,0 #on initialise la position du katamino à (0,0) et le nombre de rotations à 0
    Coups=[] #liste des coups possibles à partir de la position
    smax= 0 #initialisation du nombre de "retournements" de la pièce
    s=0
    if n-K.area<0:  #s'il n'y a plus de place sur le plateau
        return "pas de coups possibles à partir de cette position"
    if not K.symmetry:  #si la pièce n'admet pas de symétrie axiale il faudra la retourner
        smax+=1
```

ANNEXE 14/15 Résolution par backtracking

```
while s<=smax:
      while r<K.max_r:
         while K.x <= (len(Plateau)-len(K.s)):
            while K.y <= (len(Plateau[0])-len(K.s[0])):
               NP=ajout(K,Plateau,K.x,K.y)        #on ajout K au plateau dans NP, variable temporaire
               i+=1
               if type(NP) != str: #si le coup est possible
                  Coups.append(NP)
               K.y+=1
            K.y=0
            K.x+=1
         K.x=0
         K.s=tourne90(K)
         r+=1
      r=0
      K.s=retourne(K)
      s+=1
   if len(Coups)==0:
      return "pas de coups possibles à partir de cette position"
   else:
      return (Coups,n-K.area)
```

ANNEXE 15/15 Résolution par backtracking

```
def Solution(Liste, Plateau=[[0 for _ in range(Large)] for _ in range (Long)],N=Long*Large):
    """renvoie toutes les solutions possibles d'un problème
    Liste est une liste de Kataminos, Plateau une matrice et N un entier"""
    if len(Liste)==0 and N==0: #vérifie que plateau est rempli
        return [[]]
    elif len(Liste)==0: #s'il n'y a plus de kataminos à poser et que la grille #n'est pas complétée
        return 'pas une solution'
    else:
        CP=Coups_Possibles(Liste[0], Plateau, N)
        if type(CP)==str:
            return 'pas une solution'
        else:
            solutions=[]
            Coups,n=CP
            for coup in Coups:
                newSolution=Solution(Liste[1:],coup,n)  #en partant de chaque coup on cherche les solutions possibles
                for i in range(len(newSolution)):
                    if type(newSolution[i]) != str:
                        solutions.append([coup]+newSolution[i]) #on ajoute les solutions trouvées
            return solutions
```