Résoudre un puzzle simple

Xavier Dupré
http://www.xavierdupre.fr/

3 mai 2013

Résumé

C'est un cadeau que j'ai reçu à Noël sous forme de puzzle. A vrai dire, je n'ai même pas essayé de résoudre le puzzle moi-même. mais je me suis qu'en scannant les pièces du puzzle (au nombre de neuf), je devrais être en mesure d'écrire un programme pour le résoudre à ma place. Ce document décrit comment je m'y suis pris.

Table des matières

1 Description des données 1
2 Résolution 5
3 Prolongements 7
4 Programme final 7

Le puzzle en question est celui de la figure 1, il ne contient que neuf pièces, chacune d'elle est carrée et chaque côté est soit la tête, soit les jambes d'une girafe pour lesquelles il y a quatre couleurs possibles. L'appariement des pièces ne dépend que des motifs sur les bords de chaque pièce et non de leur forme. La forme finale du puzzle - un grand carré - est connue. Même s'il paraît simple, il existe un grand nombre d'arrangements possibles : il existe 9! façons possibles de placer chaque pièce et à chaque position, chaque pièce peut être tournée 4 fois. Nous oublierons volontairement les symétries car il plus difficile d'en tenir compte dans un programme informatique que de faire l'impasse dessus.

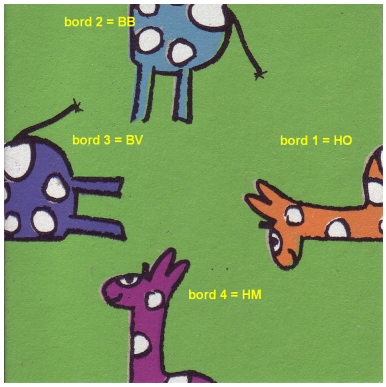
Avant de pouvoir construire la résolution du puzzle, il faut tout d'abord décrire les éléments qui le compose afin de pouvoir les manipuler. C'est l'objet du paragraphe suivant.

1 Description des données

La description des données suggère deux problèmes. Le premier est l'acquisition et le second leur représentation au sein du programme informatique. Un rapide coup d'œil sur les images et on détecte facilement qu'il existe 2x4=8 différents bords possibles et qu'une pièce peut simplement être décrite par la donnée de ces quatre bords. Mais les obtenir depuis l'image directement fait appel à des techniques de traitements d'images qui seraient trop longues à mettre en place. Il est beaucoup plus simple de construire un fichier texte dans lequel seront stockées les informations utiles concernant chaque pièce.

Il n'existe pas de manière unique de décrire un bord, le moyen envisagé est le suivant : chaque pièce va être décrite par une chaîne de huit caractères, 4x2 pour être exacte. Chaque bord est décrit par

deux lettres, la première H ou B désigne le haut ou le bas de la girafe. La seconde lettre M, O, B ou V désigne la couleur mauve, orange, bleu ou violet. Par exemple, la première pièce HOBBBVHM possède quatre bords définis comme suit :



1. HO = haut orange
2. BB = bas bleu
3. BV = bas violet
4. HM = haut mauve

Mises bout à bout, ces descriptions aboutissent au contenu de la figure 4 (page 16). Il ne reste plus qu'à écrire un petit bout de code capable de lire ce fichier puis de construire chaque pièce. Mais avant cela, il faut imaginer une représentation informatique de ces données qui va recevoir l'information lue depuis les chaînes de caractères.

Les classes ne sont pas indispensables d'autant plus que aucun héritage ne sera nécessaire dans ce cas présent. Toutefois, elles simplifient la description de chaque pièce : le puzzle comporte neuf pièces qui chacune possède quatre bords. Le programme contiendra trois classes : `Bord`, `Piece`, `Puzzle`. Ces trois classes contiendront à la fois des données et des méthodes qui s'appliquent à ces données.

1.1 classe `Bord`

La classe `Bord` ne contient que deux attributs, `partie` qui désigne la partie du corps de la girafe et `couleur`, sa couleur. Ce sont les seules données. Il n'y a pas besoin de données supplémentaires pour dire si deux bords peuvent être appariés, ce sera la méthode `compatible` et incluse dans la classe `Bord`. Deux bords pourront seront compatibles s'ils ont la même couleur et des parties opposées.

```
    def compatible (self, bord) :
        return self.couleur == bord.couleur and self.partie != bord.partie
```

Les autres méthodes de cette classe sont le constructeur `__init__` permettant de renseigner les attributs de la classe depuis une chaîne de caractères et la méthode `__str__` chargée d'afficher le bord.

1.2 classe `Piece`

Une pièce est tout d'abord composée de quatre bords, la classe `Piece` doit nécessairement contenir une liste de quatre instances de la classe `Bord`. La seule difficulté consiste à préciser à quel bord de la pièce correspond le premier bord de la liste. On décide de tourner dans le sens trigonométrique.

	bord[1]	
bord[2]	. pièce .	bord[0]
	bord[3]	

La description d'une pièce fait également intervenir sa position et son orientation (le fait de tourner la pièce de 0,90,180 ou 270 degrés) par rapport à sa description d'origine. Ce ne sont pas des informations immuables, elles seront le résultat de la résolution du puzzle mais il faut décider où placer ces informationsé : au sein de la classe `Piece` ou au sein de la classe `Puzzle`. Il n'y a pas de meilleur choix, tout dépend de la manière de rédiger l'algorithme ou de comment ces classes seront utilisées par la suite. Dans notre cas, le choix est de les insérer dans la classe `Piece`, ce seront les attributs `orientation` et `position`. Ce dernier mérite un peu plus d'attention car son type sera entier et non un t-uple de deux entiers.

La position [1] pourrait être décrite grâce à deux coordonnées entières mais le sera grâce à un seul entier. On obtient les coordonnées comme ceci d'après la position $n \in \{1, ..., 0\}$:

$$x = \{(n-1) \mod 3\} + 1 \tag{1}$$

$$y = \left[\frac{n-1}{3}\right] + 1 \tag{2}$$

Réciproquement, la position n est obtenue à partir des coordonnées $(x, y) \in \{1, ..., 3\}^2$:

$$n = (y-1) * 3 + x \tag{3}$$

La position est un entier dans l'ensemble $\{1, ...9\}$. Lors de la résolution, certaines pièces seront placées et d'autres pas encore. La convention choisie pour une pièce non encore placée est de lui affecter une position nulle. Dans ce cas, l'orientation n'a pas d'importance.

La connaissance de l'orientation et de la position de deux pièces permet également de savoir si elles peuvent être voisines étant donné le motif de leur bord conjoint s'il existe. Ce sera l'objectif de la méthode `voisin_possible` qui retourne `True` si les deux pièces font bon ménage ensemble. Le premier test porte sur les positions des pièces et de savoir si elles sont effectivement voisines. Pour cela, on calcule la différence d des positions. Les deux pièces sont voisines si d est égale à 1 ou -1 et si les pièces sont sur la même ligne ou que d est égale à 3 ou -3.

```
    def voisin_possible (self, p, a) :
        d = p.position - self.position
        if abs (d) == 1 and (p.position - 1) / 3 == (self.position - 1) / 3 :
            # voisin en x
            return True
        elif abs (d) == 3 :
            # voisin en y
            return True
        else :
            # pas voisin
            return False
```

Une fois cette condition vérifiée, il reste à vérifier que le bord qu'elles ont en commun ont des motifs appariés. Il faut donc dans un premier temps savoir quels sont les deux bords en contact. Il faut

1. La vérification des affirmations qui suivent est laissée au soin du lecteur. Le procédé est identique à celui utilisé pour la représentation des matrices par des tableaux d'une seule dimension.

d'abord les nommer, ils seront appelés bord 0, 90, 180, 270 en tournant dans le sens trigonométrique. Lorsqu'on tourne la pièce de 90 degrés, le 0 est à la place du bord 90. En tenant compte de l'orientation, on peut calculer l'indice a' qui tient compte de l'orientation o (voir la figure 2). Soit $a \in \{0, 90, 180, 270\}$ le nom du bord cherché et $o \in \{0, 90, 180, 270\}$ l'orientation de la pièce, dans ce cas :

$$a' = \frac{(a - o + 360) \mod 360}{90} \in \{0, 1, 2, 3\} \text{ est l'indice du bord cherché} \quad (4)$$

Deux pièces sont voisines si elles sont à gauche et droite l'une de l'autre (premier cas) ou en haut et bas l'une de l'autre (second cas). Le dessin de la figure 3 permet de trouver les paires d'indices a' qui correspondent aux bords en contact, et connaissant l'orientation de chaque pièce, les indices a des bords en contact. La méthode `compatible` de l'un des deux bords `dira` si effectivement les deux bords en contact sont appariés.

Les autres méthodes de la classe `Piece` concernent l'affichage et la création des pièces à partir du format décrit au paragraphe 1 et stocké dans un fichier texte (voir figure 4). Les autres attributs sont également liés à l'affichage, un nom de fichier, et deux images. Leur utilisation sera décrite au paragraphe 2.

1.3 classe `Puzzle`

Avant de construire de chercher à résoudre le problème, il faut s'interroger sur la forme du résultat que l'on cherche. Dans le cas de ce puzzle, ce peut être une liste de 9 `Piece` et pour chacune d'entres elles, les attributs `position` et `orientation` doivent être renseignés ; c'est à dire être compris en 1 et 9 pour la position et dans l'ensemble $\{0, 90, 180, 270\}$ pour l'orientation.

La classe `Puzzle` contiendra donc une liste de `Piece` précédemment décrite et incluera les méthodes nécessaires à la résolution du puzzle. La description du principe général de la résolution interviendra après celles de méthodes simples de la classe `Puzzle` [2].

piece_position

La liste résultante n'est pas forcément ordonnée en ce sens que la pièce en première position dans la liste n'est pas forcément la première pièce du puzzle. C'est l'attribut `position` de chaque pièce qui détermine la position de celle-ci. La méthode `piece_position` retourne la pièce à une certaine position donnée ou `None` si l'algorithme n'a pas encore posé de pièce à cet endroit.

```
    def piece_position (self, pos) :
        for p in self.piece :
            if p.position == pos :
                return p
        return None
```

ensemble_voisin

Cette fonction a pour objectif de retourner la liste des positions voisines à une position donnée. Le résultat est illustré par la figure 5.

angle_possible

Il s'agit ici de déterminer une orientation possible pour la pièce `p` sachant qu'elle est à la position `p.position`. Dans ce cas, on détermine l'ensemble des positions voisines à `p.position`, puis l'ensemble

2. On utilise parfois le terme de *briques élémentaires*, l'algorithme jouant le rôle de la charpente.

des pièces posées sur ces cases voisines. Il ne reste plus qu'à déterminer pour une pièce toutes les orientations qui ne dérogent pas à la règle des appariements de côtés. La fonction retourne une liste d'au plus quatre éléments et vide si une pièce ne peut être placée à la position `p.position` dans quelque orientation que ce soit.

```
    def angle_possible (self, p, display = False) :
        voisin = self.ensemble_voisin (p.position)
        if display :
            print "voisin = ", voisin
        res = []
        for a in [0,90,180,270] :
            r = True
            for v in voisin :
                piece = self.piece_position (v)
                if piece != None :
                    r = r and piece.voisin_possible (p, a)
            if r :
                res.append (a)
        return res
```

2 Résolution

La description des éléments du puzzle est terminée ainsi que quelques manipulations simples. Avant d'entamer la résolution proprement dite, il est sans doute utile de préciser la complexité du problème souvent exprimée par l'étendue de l'ensemble des solutions. Chaque pièce doit être placée à une position donnée parmi 9 dans une orientation donnée parmi 4. Avec 9! permutations, il y a donc potentiellement 9! x 4^9 = 18.874.368 solutions envisageables et pour chacune d'elles, il suffit qu'un appariement ne soit pas possible pour que la solution n'en soit pas une.

Il est possible d'explorer l'ensemble des solutions avec les ordinateurs de bureau actuels. Avec 16 pièces, ce serait encore très long (de l'ordre de 10^{23}). En théorie, l'algorithme proposé ne réduit pas forcément le nombre de solutions à explorer, ceci dépend entre autres de l'ordre initial des pièces. En pratique, il permet souvent d'éliminer des sous-ensembles de solutions. Par exemple, supposons que cinq pièces soient déjà placées aux positions 1 à 5. S'il n'est pas possible de placer aucune des autres pièces à la position 6 dans quelque orientation que ce soit, aucune des solutions avec ces cinq pièces ne marchera, soit 6144 solutions non essayées grâce à quelques tests.

La résolution du puzzle passe par la construction d'une fonction récurrente qui s'inspire de celle qu'on utilise pour construire l'ensemble des permutations de $\{1, ..., n\}$.

2.1 Permutations

Il est assez intuitif de construire par récurrence l'ensemble des permutations de $\{1, ..., n\}$ à l'instar de la fonction factorielle. Parmi les $n!$ permutations, il y a extactement $(n-1)!$ permutations avec le nombre 1 en première position, il y a aussi $(n-1)!$ avec le nombre 2 en première position et ainsi de suite. En s'aidant de cette constatation, on cherche à construire la fonction `permutation(n)` qui retourne la liste des permutations de l'ensemble $\{1, ..., n\}$ par récurrence.

On note P_n les permutations de l'ensemble $\{1, ..., n\}$. Les premières $(n-1)!$ permutations sont celles où le premier nombre 1 est fixé en première position et où on applique les permutations P_{n-1} à l'ensemble restant $\{2, ..., n\}$. Les permutations suivantes sont celles où on permute 1 avec 2 et où on applique les permutations à l'ensemble $\{1, 3, 4, ..., n\}$. On obtient en continuant ce raisonnement :

première vague de permutations : 1 + permutations appliquées à $\{2, 3, 4, 5, ..., n\}$
seconde vague de permutations : 2 + permutations appliquées à $\{\mathbf{1}, 3, 4, 5, ..., n\}$
troisième vague de permutations : 3 + permutations appliquées à $\{2, \mathbf{1}, 4, 5..., n\}$
quatrième vague de permutations : 4 + permutations appliquées à $\{2, 3, \mathbf{1}, 5..., n\}$
...

Le programme complet aboutit à :

```
# coding: latin-1
def echange (l, i, j) :
    """permutation des éléments i et j de la liste l"""
    t       = l [i]
    l [i]   = l [j]
    l [j]   = t

def applique_permutation (set, p) :
    """applique la permutation p à l'ensemble set"""
    return [ set [k-1] for k in p ]

def permutation (n) :
    if n == 1 : return [ [ 1 ] ]
    else :
        per = permutation (n-1)             # permutation sur {1,...,n-1}
        set = [ i+1 for i in range (0,n) ]  # ensemble de départ
        res = []                            # résultat

        for i in range (0, n) :
            echange (set, 0, i)             # échange élément 0 et i
            for p in per :                  # boucle sur les permutations de {1,...,n-1}
                pr  = [ set [0] ] + applique_permutation ( set [1:], p )
                res.append (pr)             # on l'ajoute au résultat
            echange (set, 0, i)             # échange élément 0 et i

        return res

print (permutation (1)) # retourne [ [ 1 ] ]
print (permutation (2)) # retourne [[1, 2], [2, 1]]
print (permutation (3)) # retourne [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1], [3, 1, 2]]
```

2.2 Application au puzzle

Il serait simple d'utiliser le programme précédent pour construire l'ensemble des permutations puis de le parcourir en changeant pour chacune d'entre elles les orientations de chaque pièce mais ce système nous ferait évaluer l'ensemble des solutions possibles sans possibilité de le réduire comme suggéré au paragraphe 2. Il vaut mieux reprendre le programme précédent et remplacer la fonction `permutation` de sorte qu'elle explore les solutions au lieu de les retourner comme résultat.

Le principe de la résolution repose sur la récurrence de la méthode `solution`. Il y a 4x9 possibilités pour le choix en première position. La méthode `solution` en choisit une puis appelle la méthode `solution` sur la seconde position, si celle-ci retourne `True` alors ce premier choix était le bon, sinon on passe au choix suivant. Au second appel, une pièce est choisie pour la seconde position puis on appelle encore la méthode `solution`. Le schéma est le suivant :

1. On calcule la liste des choix pour la position p.

2. Si cet ensemble est vide, on retourne `False`, il n'y a pas de solution possible dans cette voie.
3. Sinon on choisit la première possibilité.
4. On appelle la méthode `solution` pour la position $p + 1$.
5. Si cette méthode retourne `True`, une solution a été trouvée.
6. Sinon, on passe au choix suivant jusqu'à ce qu'il n'y en ait plus.

L'implémentation proposée ne suit pas tout-à-fait ce schéma puisqu'on ne calcule pas l'ensemble des choix possibles au début de la méthode `solution` mais on calcule un premier choix (méthode `meilleur_choix` en tenant des compte des pièces déjà placées (dans la liste `free`). Si la solution finale utilise cette pièce, c'est terminé sinon, la fonction `meilleur_choix` est appelée une seconde fois pour obtenir un second choix de pièce en tenant compte des pièces placées et essayées. Le paragraphe 4 reprend le code de la méthode `solution`. La figure 8 montre le puzzle fini.

Il convient de rapprocher l'utilisation de méthodes récursives au concept de graphe ou plus précisément d'arbre. La figure 6 illustre ce que peut être un tel arbre dans le cas de ce puzzle. Chaque nœud représente un nouvel appel à la méthode `solution`. Chaque nœud est relié par un arc au nœud précédent l'ayant appelé. Une branche (depuis la racine jusqu'à une feuille) longue de moins de 9 nœuds signifie qu'une contradiction est apparue avant de placer toutes les pièces, ceci signifie qu'il n'était pas nécessaire d'explorer toutes les branches passant par les derniers nœuds intermédiaires. Une branche de 9 nœuds signifie que la contradition est apparue sur la dernière pièce et une seule branche de 9 nœuds a mené à la solution trouvée.

3 Prolongements

On retrouve des puzzles carrés comme celui-là avec des photos aériennes (voir figure 7). Une zone géographique a été balayée avec un avion ou un satellite et on dispose d'une multitude de photos carrées disjointes recouvrant la zone. Le nombre de pièces est bien souvent plus grand que 9, une autre difficulté est que parfois, on ne connaît pas les dimensions du puzzle final et il faut trouver le meilleur arrangement. Enfin, autant dans l'exemple des girafe, il est facile de dire si deux côtés sont appariés ou non, autant avec des photos aériennes, ce n'est pas toujours un problème facile. Cela est dû au fait que les photos ne sont pas toujours disjointes, que certaines photos ne sont pas prises dans les mêmes conditions (un nuage passe et les couleurs sont assombries), la compression à une faible résolution puis la décompression déteriore la qualité des images. Dans ces conditions, la notion d'appariement est plus une notion floue.

4 Programme final

Le programme s'ouvre sur les neuf pièces disposées à une position correspondant à leur ordre d'apparition et dans leur orientation d'origine. Un clic de souris fait démarrer le programme. La recherche de la solution fait apparaître les choix effectués par l'algorithme jusqu'à ce qu'une solution se dégage. Un dernier clic de souris termine le programme. La fonction `sleep` permet de ralentir l'exécution du programme et de rendre le défilement des pièces humainement visible.

Les images sont chargées une fois puis recopiées et réduites. Une image de petite taille signifie que la pièce n'a pas encore été placée par la méthode de résolution.

```
# coding: cp1252
import pygame
```

```
import random
import copy
import time

class Bord :
    """définition d'un bord ou côté d'une pièce, il possède :
       - partie : une partie de la girafe (haut ou bas)
       - une couleur : la couleur de cette partie, (orange, violet, bleu clair, bleu foncé)
    """

    def __init__ (self, definition) :
        """definition est une chaîne de 2 caractères qui définit un bord, exemple :
            HO pour haut orange
            BB pour bas bleu
            BV pour bas violet
            HM pour haut mauve
        """

        if   definition [0] == "H" : self.partie = "haut"
        elif definition [0] == "B" : self.partie = "bas"
        else :                       self.partie = definition + "###"

        if   definition [1] == "O" : self.couleur = "orange"
        elif definition [1] == "B" : self.couleur = "bleu clair"
        elif definition [1] == "M" : self.couleur = "violet"
        elif definition [1] == "V" : self.couleur = "bleu fonce"
        else :                       self.couleur = definition + "##"

    def __str__ (self) :
        """cette méthode est appelée lorsqu'on exécute l'instruction print
        avec un objet de type Bord"""
        s = self.partie + " " + self.couleur
        return s

    def compatible (self, bord) :
        """dit si deux bords sont compatibles, c'est à dire
        de la même couleur et de partie différente"""
        return self.couleur == bord.couleur and self.partie != bord.partie


class Piece :
    """définition d'une pièce du puzzle, celle-ci inclut :
      - bord : cette liste contient quatre objets de type Bord, cette liste ne changera plus
      - position    : c'est la position de la pièce dans le puzzle, ce qui nous intéresse,
                      c'est la position finale de la pièce dans le puzzle, cette information
                      va donc bouger au fur et à mesure que nous allons essayer de
                      résoudre le puzzle
      - orientation : de même que pour la position, une pièce peut être tournée sans changer de
                      position, c'est le résultat final qui nous intéresse

    pour l'affichage, on ajoute deux informations :
      - name  : le nom de l'image de la pièce
      - image : c'est la représentation de l'image dans la mémoire de
                l'ordinateur pour le module pygame
    """

    def __init__ (self, name, definition, position, numero) :
        """ on définit la pièce
           - name : nom de l'image représentant la pièce
```

```
        - definition : chaîne de 8 caractères, c'est une suite de 4 x 2 caractères définissant
                       chaque bord, voir la classe bord pour leur signification
        - position   : c'est la position initiale de la pièce, on suppose que
                       l'orientation est nulle pour commencer

    à partir de ces informations, on construit :
      - image       : c'est la représentation en mémoire de l'image de la pièce
      - bord        : c'est une liste qui définit les 4 bords de la pièce
      - orientation : c'est l'orientation de la pièce, au début de la résolution, elle est nulle
    """

    self.name           = name
    image               = pygame.image.load (name)
    self.image          = pygame.transform.scale (image, (250,250))
    s                   = self.image.get_size ()
    self.image_petite   = pygame.transform.scale (self.image,  (int(s[0]*0.7), int(s[1]*0.7)))
    self.bord           = []

    for i in range (0,4) :
        self.bord.append ( Bord (definition [i*2:i*2+2] ) )

    self.orientation    = 0
    self.position       = position
    self.numero         = numero

def __str__ (self) :
    """définition ce qu'on doit afficher lorsqu'on exécute
    l'instruction print avec un objet de type Piece"""
    s = str (self.position) + " : "
    for b in self.bord :
        s += str (b) + " - "
    s += " orientation " + str (self.orientation)
    s += " numero " + str (self.numero)
    return s

def affiche (self, screen, position) :
    """affiche la pièce à l'écran en tenant compte de sa position et de son orientation"""
    if "darker" in self.__dict__ and self.darker :
        position = ( position [0] + 20, position [1] + 20 )
        image    = pygame.transform.rotate (self.image_petite, self.orientation)
        screen.blit (image, position)
    else :
        image = pygame.transform.rotate (self.image, self.orientation)
        screen.blit (image, position)

def bord_angle (self, angle, orientation = None) :
    """retourne le bord connaissant l'orientation de la pièce,
    le bord demandé est celui correspondant à :
       - 0  bord droit
       - 90 bord haut
       - 180 bord gauche
       - 270 bord bas"""
    if orientation == None : return self.bord_angle (angle, self.orientation)
    else :
        dif = (angle - orientation + 360) % 360 / 90
        return self.bord [dif]

def voisin_possible (self, p, a) :
    """détermine si la pièce self peut être voisine avec la pièce p tournée de l'angle a"""
```

```
        d = p.position - self.position
        if abs (d) == 1 and (p.position - 1) / 3 == (self.position - 1) / 3 :
            # voisin en x
            if d == 1 :
                a1 = 0
                a2 = a1 + 180
            else :
                a1 = 180
                a2 = 0
        elif abs (d) == 3 :
            # voisin en y
            if d == 1 :
                a1 = 90
                a2 = 270
            else :
                a1 = 270
                a2 = 90
        else :
            # pas voisin
            return False

        b1 = self.bord_angle (a1)
        b2 = p.bord_angle (a2, a)
        return b1.compatible (b2)


class Puzzle :
    """définition d'une classe puzzle, elle contient simplement
    une liste de 9 pièces dont les positions sont :
    1 2 3
    4 5 6
    7 8 9
    et les orientations choisies dans l'ensemble { 0,90,180,270 }
    """


    def __init__ (self, dir = ".") :
        """on définit le puzzle à partir des informations contenus
        dans le répertoire dir qui doit contenir :
        - 9 images appelées piece0.png, ..., piece1.png
        - un fichier definition.txt contenant la définition de
          chacun des 4 bords de chacune des 9 pièces
                            HOBBBVHM
                            HOBBBVHM
                            HBBMBVHO
                            HMBBBVHB
                            BMBOHBHV
                            HVBMBOHM
                            BMBVHBHO
                            HVHMBBBO
                            BMHOHVBB
        """

        # on lit définition.txt
        f = open (dir + "/definition.txt")
        bo = f.readlines ()
        f.close ()

        # on définit chaque pièce
```

```
        self.piece = []
        for i in range (1,10) :
            name    = dir + "/piece" + str (i) + ".png"
            d       = bo [i-1]
            p       = Piece (name, d, 0, i)
            self.piece.append ( p )

    def __str__ (self) :
        """ce qu'on doit afficher lorsqu'on exécute
        l'instruction print avec un objet de type Puzzle"""
        s = """1 2 3
4 5 6
7 8 9
"""
        for p in self.piece :
            s += str (p) + "\n"
        return s

    def pixel (self, position) :
        """retourne en fonction de la position (1 à 9) de la pièce sa position sur l'écran,
        soit deux coordonnées"""
        p       = position - 1
        ligne   = p / 3
        colonne = p % 3
        return (colonne * 250, ligne * 250)

    def affiche (self, screen, petite = False) :
        """affiche les pièces sur l'écran,
        en plus petit pour celles qui ne sont pas encore placées"""
        screen.fill ((0,0,0))
        free = [0 for i in self.piece]
        for p in self.piece :
            if p.position > 0 :
                p.darker = False
                p.affiche (screen, self.pixel (p.position))
                free [p.position-1] = 1

        if petite :
            em = []
            for i in range (0, len (free)) :
                if free [i] == 0 : em.append (i+1)
            i = 0
            for p in self.piece :
                if p.position == 0 :
                    p.darker = True
                    p.affiche (screen, self.pixel (em [i]))
                    i += 1

        pygame.display.flip ()

    def meilleure_piece (self, free, pos) :
        """retourne la prochaine pièce à placer sur le puzzle,
        dans un premier temps, on peut prend la première qui vient,
        ensuite, on peut essayer un choix plus judicieux"""
        if len (free) == 0 : return None
        else : return free [0]

    def piece_position (self, pi) :
        """recherche la piece associée à la position pi"""
```

```
        for p in self.piece :
            if p.position == pi :
                return p
        return None

    def ensemble_voisin (self, i) :
        """retourne les positions voisins de la position i"""
        i -= 1
        res = []
        for x in [-1,0,1] :
            for y in [-1,0,1] :
                if abs (x) == abs (y) : continue
                if x == -1 and i % 3 == 0 : continue
                if x == 1  and i % 3 == 2 : continue
                if y == -1 and i / 3 == 0 : continue
                if y == 1  and i / 3 == 2 : continue
                j = i + x + y * 3
                if j in range (0,9) :
                    res.append (j)
        return [ j+1 for j in res ]

    def nb_place (self) :
        """retourne le nombre de places vides"""
        i = 0
        for p in self.piece :
            if p.position == 0 : i += 1
        return i

    def angle_possible (self, p, display = False) :
        """retourne l'ensemble des angles possibles pour une pièce donnée"""
        voisin = self.ensemble_voisin (p.position)
        if display :
            print "voisin = ", voisin
        res = []
        for a in [0,90,180,270] :
            r = True
            for v in voisin :
                piece = self.piece_position (v)
                if piece != None :
                    r = r and piece.voisin_possible (p, a)
            if r :
                res.append (a)
        return res

    def solution (self, pos = 1, screen = None) :
        if pos == 1 :
            for p in self.piece :
                p.position = 0
            self.nb_position = 0
            self.essai = 0

        self.essai += 1

        if self.nb_position == len (self.piece) :
            time.sleep (0.2)
            return

        # le tableau free mémorise l'ensemble des pièces non encore placées
        free = []
```

```
        for p in self.piece :
            if p.position == 0 :
                free.append (p)

        if screen != None :
            self.affiche (screen, True)
            pygame.display.flip ()

        p = self.meilleure_piece (free, pos)
        # si p == None, on n'étudie pas les solutions avec les pièces placées
        # avant aux positions 1 à pos --> on n'entrera pas dans la boucle suivante
        while p != None :

            p.position     = pos
            angle          = self.angle_possible (p)

            # p est la pièce choisie, pour cette pièce
            # on passe en revue toutes les orientations
            for a in angle :
                p.orientation = a

                time.sleep (0.2)  # temporisation

                if self.nb_place () == 0 :  return True
                else :
                    r = self.solution (pos+1, screen = screen)
                    if r : return True

            p.position     = 0

            # on réactualise le tableau free qui aura été modifié par
            # l'appel à self.solution et on enlève le choix précédemment testé
            free2 = free
            free  = []
            for f in free2 :
                if f.numero != p.numero : free.append (f)

            # on passe au choix suivant avec free contenant les pièces
            # placées et les pièces essayées
            p = self.meilleure_piece (free, pos)


def attendre_clic ():
    """attend la pression d'un clic de souris"""
    reste = True
    while reste:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONUP :
                reste = False
                break

if __name__ == "__main__" :

    # taille de l'écran
    size = (750,750)

    # initialisation du module pygame
    pygame.init ()
    screen  = pygame.display.set_mode (size)
```

```
# on définit le puzzle
p = Puzzle ()

# on affiche le puzzle avec print (format texte)
print p

# on affiche le puzzle à l'écran
p.affiche (screen, petite = True)
attendre_clic ()

# on rafraîchit l'écran pour que le puzzle apparaissent
pygame.display.flip ()

# on trouve la solution
r = p.solution (screen = screen)
print "résolution ", r
print "appelle à la méthode solution ", p.essai

# on attend la pression d'un clic de souris avant de terminer le programme
p.affiche (screen, petite = True)
attendre_clic ()
```

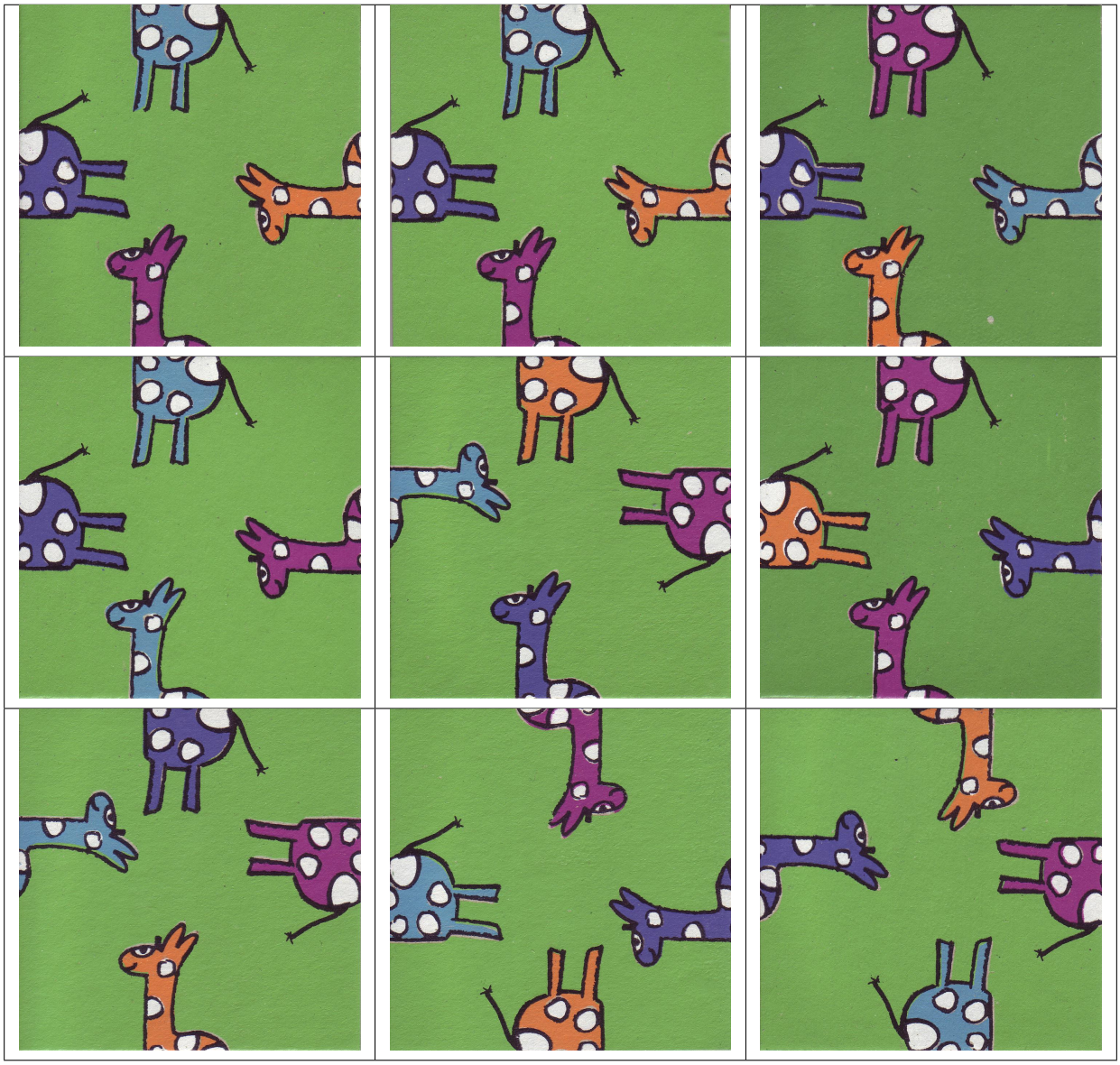


Figure 1 : *Les neuf pièces du puzzle (voir la figure 8 pour les voir assemblées).*

	$0 = 90 + 90$ $a' = a - o$	
$90 = 180 + 90$ $a' = a - o$	pièce tournée de 90 degré	$270 = 360 + 90$ $a' = a - o$
	$180 = 270 + 90$ $a' = a - o$	

Figure 2 : *Correspondance entre a, o, a' lorsqu'une pièce est tournée de 90 degré dans le sens trigonométrique.*

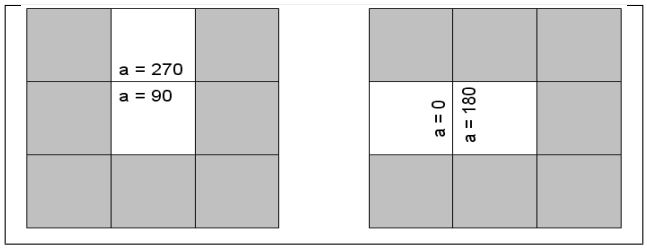


Figure 3 : *Deux pièces sont voisines en x ou en y. Les bords voisins sont dans le premier cas* 0 *et* 180*, dans le second,* 90 *et* 270 *(indices a').*

```
HOBBBVHM
HOBBBVHM
HBBMBVHO
HMBBBVHB
BMBOHBHV
HVBMBOHM
BMBVHBHO
HVHMBBBO
BMHOHVBB
```

Figure 4 : *Description des neuf pièces du puzzle au format texte, une pièce par ligne. Chaque ligne est une séquence de quatre groupes de deux lettres. Chaque groupe désigne un bord d'une pièce. La première lettre (H ou B) désigne le haut ou le bas d'une girafe. La seconde lettre (O, M, V, B) désigne sa couleur (orange, mauve, violet, bleu).*

1 2,4	**2** 1,3,5	**3** 2,6
4 1,5,7	**5** 2,4,6,8	**6** 3,5,9
7 4,8	**8** 5,7,9	**9** 6,8

```
def ensemble_voisin (self, i) :
    i -= 1
    res = []
    for x in [-1,0,1] :
        for y in [-1,0,1] :
            if abs (x) == abs (y) : continue
            if x == -1 and i % 3 == 0 : continue
            if x == 1  and i % 3 == 2 : continue
            if y == -1 and i / 3 == 0 : continue
            if y == 1  and i / 3 == 2 : continue
            j = i + x + y * 3
            if j in range (0,9) :
                res.append (j)
    return [ j+1 for j in res ]
```

Figure 5 : *Ensemble des positions voisines pour chacune des positions.*

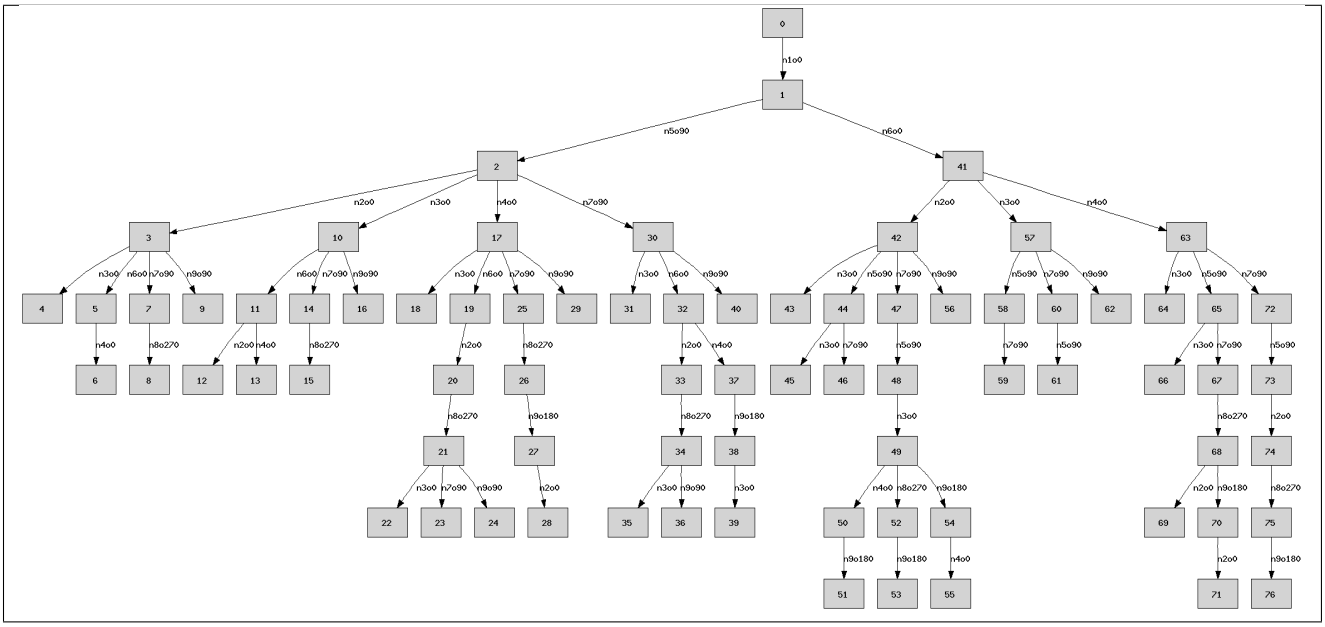


Figure 6 : *Dépliement des appels récursifs de la méthode* `solution` *sous forme d'arbre. Le numéro de chaque nœud désigne le* $n^{ième}$ *appel à la méthode* `solution`*. Sur chaque arc est superposé un label du type* `n3o90`*, ceci signifie que l'algorithme a placé la pièce 3 tournée de 90 degrés à une position indiquée par la profondeur du nœud dans l'arbre.*

Figure 7 : *Prises de vue aériennes recouvrant une zone géographique. Le puzzle reconstruit n'a pas une forme déterminée à l'avance.*

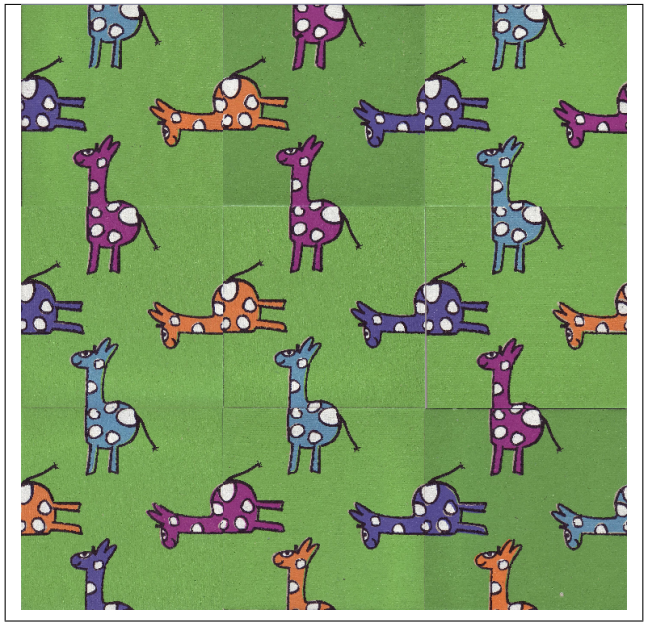


Figure 8 : *Le puzzle fini.*