Python (M1) – Des classes, des points et des arbres

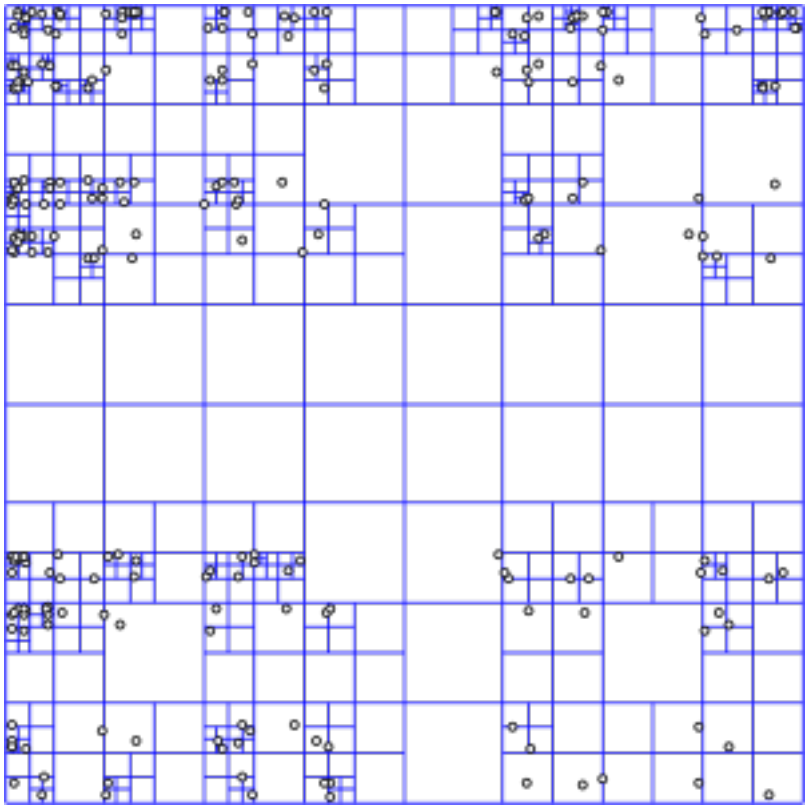
TD #5

Rédacteur: Stéphane Vialette

1 Arbres quaternaires

1.1 Introduction

L'*arbre quaternaire* (*quadtree* en anglais) est une structure hiérarchique construite par divisions récursives de l'espace en quatre quadrants disjoints.



Pour être représentée par un arbre quaternaire, un ensemble de points (vu comme une image dans la suite) est récursivement décomposée en quatre quadrants disjoints de même taille de telle sorte que chaque nœud de l'arbre quaternaire représente un quadrant de l'image. Le nœud racine de l'arbre représente l'image entière. Un nœud est une *feuille* si le quadrant correspondant dans l'image ne contient qu'un seul point, sinon le nœud est *interne*.

1.2 Un peu plus de détails

Nous nous intéresserons dans ce TP à un arbre quaternaire un peu particulier. Celui-ci sera en effet paramétré par sa *granularité*, c'est-à-dire le nombre maximum de points stockés dans une feuille. Ainsi, si dans un arbre quaternaire classique une feuille stocke exactement un point, les arbres quaternaires considérés ici permettrons de stocker plusieurs points.

Pour construire un arbre quaternaire, nous aurons donc besoin d'un ensemble de points et de la granularité souhaitée pour l'arbre construit. L'algorithme de construction est le suivant :

- Si le nombre de points est inférieur ou égal à la granularité souhaitée, créer une feuille et stocker les points dans cette feuille.
- Si le nombre de points est supérieur à la granularité souhaitée, décomposer l'image courante (un carré) est quatre quadrants disjoints de même taille (quatre carrés) et construire récursivement l'arbre quaternaire en filtrant les points relativement à chacun des quadrants.

L'image (le carré) courante sera également stockée dans chaque nœud de l'arbre quaternaire (qu'il s'agisse d'une feuille ou d'un nœud interne). Autrement dit chaque sommet u de l'arbre stocke la partie du plan dans laquelle se trouve tous les points stockés dans les feuilles du sous-arbre enraciné en u.

Les arbres quaternaires ont de nombreuses applications en traitement d'image et en géométrie algorithmique. Nous verrons dans la dernière partie de ce TP comment utiliser des arbres quaternaires pour rechercher efficacement un sous-ensemble de points appartenant à un rectangle requête.

2 Des points et des rectangles

Il est temps d'entrer dans le vif du sujet. Cependant avant de considérer l'implémentation des arbres quaternaires, nous allons nous intéresser à deux points prélimaires : comment représenter des points et des rectangle de l'espace.

2.1 Les points du plan

Nous utiliserons la classe (minimale) `Point` définie dans le module `point`. Un point sera représenté par sa coordonnée `x` et par sa coordonnée `y`. Ces objets sont non mutables, il n'est pas possible de modifier les coordonnées d'un point (actions réalisées par propriétés python).

```
##############################################################################

class Point(object):
    '''
    A simple 2D point class
    '''
```

```
    def __init__(self, x = 0, y = 0):
        '''
        Initialize a point with a and y coordinates
        '''
        self._x = x
        self._y = y

    def __eq__(self, point):
        '''
        Return True iff this point and the other point have the same coordinates
        '''
        return (self._x == point._x and self._y == point._y)

    def __str__(self):
        '''
        Stringify this point
        '''
        return '%s(%f,%f)' % (self.__class__.__name__, self._x, self._y)

    def get_x(self):
        '''
        Return the x coordinate of this point
        '''
        return self._x

    x = property(get_x, None, None, 'x coordinate read only property')

    def get_y(self):
        '''
        Return the y coordinate of this point
        '''
        return self._y

    y = property(get_y, None, None, 'y coordinate read only property')

    def dominates(self, point):
        '''
        Return True iff this point dominates the given point
        '''
        return self._x > point._x and self._y > point.y

################################################################################
```

Le module `testpoint` contient quelques tests unitaires permettant de tester la classe `Point`.

2.2 Des rectangles

Considérons la classe `Rectangle` définie dans le module `rectangle`. Un rectangle (cotés parallèles aux axes) sera défini par son point extrémité inférieur gauche et par son point extrémité supérieur droit. Encore une fois ces objets sont non mutables, il n'est pas possible de modifier un rectangle une fois celui-ci créé.

```
################################################################################

class RectangleException(Exception):
    pass


################################################################################


class Rectangle(object):
    '''
    A axis-parallel rectangle class defined by
    a lower left point and an upper right point
    '''

    def __init__(self, lower_left_point, upper_right_point):
        '''
        Initialize a rectangle from lower left and upper right points
        '''
        if not upper_right_point.dominates(lower_left_point):
            raise RectangleException('Bad coordinates')
        self._lower_left_point = lower_left_point
        self._upper_right_point = upper_right_point

    def __str__(self):
        '''
        Stringify a rectangle
        '''
        return '%s(%s,%s)' % \
            (self.__class__.__name__,
             str(self._lower_left_point),
             str(self._upper_right_point))

    def __contains__(self, point):
        '''
        Return True iff a given point lies inside this rectangle (p in r)
        '''
        pass # A COMPLETER

    def contains_rectangle(self, rectangle):
        '''
        Return True iff a given rectangle lies completely inside this rectangle
        '''
        pass # A COMPLETER

    def does_not_intersect_with_rectangle(self, rectangle):
        '''
        Return True iff a given rectangle does not intersect with this rectangle
        '''
        if self._lower_left_point.x > rectangle._upper_right_point.x:
            # This rectangle is strictly to the right of the given rectangle
            return True
```

```
        elif self._upper_right_point.x < rectangle._lower_left_point.x:
            # This rectangle is strictly to the left of the given rectangle
            return True
        elif self._upper_right_point.y < self._lower_left_point.y:
            # This rectangle is strictly below the given rectangle
            return True
        elif self._lower_left_point.y > self._upper_right_point.y:
            # This rectangle is strictly above the given rectangle
            return True
        # still here !?
        return False

    def intersects_with_rectangle(self, rectangle):
        '''
        Return True iff a given rectangle intersects with this rectangle
        '''
        return not self.does_not_intersect_with_rectangle(rectangle)

    def get_lower_left_point(self):
        '''
        Return the lower point of this rectangle
        '''
        return self._lower_left_point

    lower_left_point = property(get_lower_left_point, None, None,
                                'lower left point read only property')

    def get_upper_right_point(self):
        '''
        Return the upper right point of this rectangle
        '''
        return self._upper_right_point

    upper_right_point = property(get_upper_right_point, None, None,
                                 'upper right point read only property')

    def width(self):
        '''
        Return the width of this rectangle
        '''
        return self._upper_right_point.x - self._lower_left_point.x

    def height(self):
        '''
        Return the height of this rectangle
        '''
        return self._upper_right_point.y - self._lower_left_point.y

################################################################################
```

Le module `testrectangle` contient quelques tests unitaires permettant de tester la classe `Rectangle`.

QUESTION 1
Écrire les méthodes `__contains__` et `contains_rectangle`.

3 Les arbres quaternaires

Nous passons à l'implémentation des arbres quaternaires (tout se déroule dans le module `quadtree`) et définissons pour cela plusieurs classes. La classe `QuadTree` est un *frontal* : elle ne contient en fait qu'une référence sur la réelle racine de l'arbre quaternaire (une instance de la classe `_QuadTreeInternalNode` ou `_QuadTreeLeaf`) et son rôle se borne à propager à la racine de l'arbre les appels à ses méthodes. Pour masquer l'implémentation, les classes `_QuadTreeInternalNode` et `_QuadTreeLeaf` sont des classes internes à la classe `QuadTree` (elles ont en fait des classes dérivées de la classe abstraite `_QuadTreeNode` ...tout nœud stocke en effet la partie du plan sur laquelle il opère).

```
################################################################################


from point import Point
from rectangle import Rectangle


################################################################################


class QuadTreeException(Exception):
    pass


################################################################################


class QuadTree(object):

    class _QuadTreeNode(object):
        '''
        Astract base class for quastree nodes
        '''

        def __init__(self, square):
            '''
            any quadtree node must store a square
            '''
            if self.__class__ is QuadTree._QuadTreeNode:
                raise QuadTreeException('QuadTree._QuadtreeNode is an abstract class')
            self._square = square

        def __iter__(self):
            '''
            Return an iterator over all points stored in the subtree rooted
```

```
            at this node.
            '''
            return iter(self._report_all_points())

        def size(self):
            '''
            To be implemented in derived class
            '''
            raise NotImplementedException('not implemented')

        def _report_all_points(self):
            '''
            To be implemented in derived class
            '''
            raise NotImplementedException('not implemented')

        def _report_points_in_rectangle(self):
            '''
            To be implemented in derived
            '''
            raise NotImplementedException('not implemented')

        def get_square(self):
            '''
            Return the square associated to the subtree rooted at this node
            '''
            return self._square

        square = property(get_square, None, None, 'square read only property')

    class _QuadTreeInternalNode(_QuadTreeNode):
        '''
        A quadtree internal node, i.e., a quadtree node with 4 children
        '''

        def __init__(self, square, upper_right, upper_left, lower_left, lower_right):
            '''
            Initalize this internal quadtree nodes
            '''
            super(QuadTree._QuadTreeInternalNode, self).__init__(square)
            self._upper_right = upper_right
            self._upper_left  = upper_left
            self._lower_left  = lower_left
            self._lower_right = lower_right

        def __str__(self):
            '''
            Stringify this quadtree internal node
            '''
            return 'QuadTreeInternalNode(%s,%s,%s,%s)' % \
             (str(self._upper_right), str(self._upper_left),
              str(self._lower_left), str(self._lower_right))
```

```
    def size(self):
        '''
        Return the number of points stored in the subtree rooted at this node
        '''
        return  self._upper_right.size() + \
            self._upper_left.size()      + \
            self._lower_left.size()      + \
            self._lower_right.size()

    def _report_all_points(self):
        '''
        Report all points stored in the subtree rooted at this node
        '''
             pass # A COMPLETER

    def _report_points_in_rectangle(self, rectangle):
        '''
        Report all points stored in the subtree rooted at this node that are
        in a query rectangle
        '''
        pass # A COMPLETER

class _QuadTreeLeaf(_QuadTreeNode):
    '''
    A quadtree leaf
    '''

    def __init__(self, square, points):
        '''
        Initialize this quadtree leaf
        '''
        super(QuadTree._QuadTreeLeaf, self).__init__(square)
        self._points = points[:]

    def __str__(self):
        '''
        Stringify this quadtree leaf
        '''
        return 'QuadTreeLeaf([%s])' % \
                              (','.join([str(point) for point in self._points]),)

    def _report_all_points(self):
        '''
        Report all points that lie in this leaf
        '''
        pass # A COMPLETER

    def _report_points_in_rectangle(self, rectangle):
        '''
        Report all points stored in this quadtree leaf that are
        in a query rectangle
        '''
        pass # A COMPLETER
```

```
def __init__(self, points, granularity):
    '''
    Initialize this quadtree
    '''
    if granularity < 1:
        raise QuadTreeException('QuadtTree requieres a positive granularity')
    if points == []:
        self._root = None
    else:
        square = QuadTree._minimum_enclosing_square(points)
        self._root = QuadTree._construct_from_points(points, square, granularity)

def __iter__(self):
    '''
    Return an iterator over all points stored in this quadtree
    '''
    return iter(self._root) if self._root is not None else iter([])

def size(self):
    '''
    Return the number of points stored in this quadtree
    '''
    return self._root.size() if self._root is not None else 0

def report_points_in_rectangle(self, rectangle):
    '''
    Return all points stored in this quadtree that are in the query rectangle
    '''
    return self._root._report_points_in_rectangle(rectangle) \
        if self._root is not None else []

@staticmethod
def _minimum_enclosing_square(points):
    '''
    Return the minimum square that contains the given points
    '''
    # x coordinate
    x_min = min(point.x for point in points)
    x_max = max(point.x for point in points)
    width = x_max - x_min
    # y coordinate
    y_min = min(point.y for point in points)
    y_max = max(point.y for point in points)
    height = x_max - x_min
    # contruct the minimum size square
    return Rectangle(Point(x_min, y_min),
                     Point(x_min + max(width, height), y_min + max(width, height)))

@staticmethod
def _construct_from_points(points, square, granularity):
    '''
    Construct a quadtree from a colection of points
```

```
        '''
        if len(points) <= granularity:
            return QuadTree._QuadTreeLeaf(square, points)

        else:
            # cut the square area into 4 square areas
            x_mid = square.lower_left_point.x + \
                ((square.upper_right_point.x - square.lower_left_point.x) / 2.0)
            y_mid = square.lower_left_point.y + \
                ((square.upper_right_point.y - square.lower_left_point.y) / 2.0)

            # upper right square area
            upper_right_square = Rectangle(Point(x_mid, y_mid),
                                           square.upper_right_point)
            points_in_upper_right_square = [point for point in points
                                            if point.x > x_mid and point.y > y_mid]
            upper_right = QuadTree._construct_from_points(points_in_upper_right_square,
                                                          upper_right_square,
                                                          granularity)

            # upper left square area
            upper_left_square = Rectangle(Point(square.lower_left_point.x, y_mid),
                                          Point(x_mid, square.upper_right_point.y))
            points_in_upper_left_square = [point for point in points
                                           if point.x <= x_mid and point.y > y_mid]
            upper_left = QuadTree._construct_from_points(points_in_upper_left_square,
                                                         upper_left_square,
                                                         granularity)

            # lower left square area
            lower_left_square = Rectangle(square.lower_left_point,
                                          Point(x_mid, y_mid))
            points_in_lower_left_square = [point for point in points
                                           if point.x <= x_mid and point.y <= y_mid]
            lower_left = QuadTree._construct_from_points(points_in_lower_left_square,
                                                         lower_left_square,
                                                         granularity)

            # lower right square area
            lower_right_square = Rectangle(Point(x_mid, square.lower_left_point.y),
                                           Point(square.upper_right_point.x, y_mid))
            points_in_lower_right_square = [point for point in points
                                            if point.x > x_mid and point.y <= y_mid]
            lower_right = QuadTree._construct_from_points(points_in_lower_right_square,
                                                          lower_right_square,
                                                          granularity)

        # done (leaf or recursive construction), construct and return the quadtree
        return QuadTree._QuadTreeInternalNode(square,
                                              upper_right,
                                              upper_left,
                                              lower_left,
                                              lower_right)
```

```
#############################################################################
```

QUESTION 2

Écrire les méthodes `_report_all_points` des classes `_QuadTreeInternalNode` et `_QuadTreeLeaf`.

QUESTION 3

Écrire les méthodes `_report_points_in_rectangle` des classes `_QuadTreeInternalNode` et `_QuadTreeLeaf`. Vous remarquerez que pour tout sommet u de l'arbre quaternaire, si la partie du plan (carré) associée à u n'intersecte pas le rectangle requête, alors aucun point stocké dans le sous-arbre enraciné en u ne peut être dans la solution. De même, si la partie du plan (carré) associée à u est complètement incluse dans le rectangle requête, alors tout stocké dans le sous-arbre enraciné en u fait partie de la solution (c'est le but des méthodes `_report_all_points` !).

4 Pour aller plus loin que ce TP

- Permettre l'ajout d'un ou plusieurs points dans un arbre quaternaire (tout en respectant la granularité de l'arbre . . .une feuille peut donc devenir une nœud interne).
- Permettre de modifier la granularité d'un arbre quaternaire.
- Supprimer un point d'un arbre quaternaire (encore une fois, la difficulté réside dans le fait qu'il faut respecter la granularité de l'arbre).