EI1022/MT1022 - Algoritmia

Entregable 4 - Fecha de entrega al aula virtual 7 de diciembre de 2018

En muchos problemas prácticos se plantea la denominada "búsqueda del vecino más próximo": dado un conjunto $S \subseteq \mathbb{R}^m$ de puntos y dado un punto $p \in \mathbb{R}^m$ (no necesariamente de S), encontrar el punto de S más cercano a p. Nosotros vamos a ocuparnos de este problema para puntos en el plano ($m = 2$). Una búsqueda exhaustiva (es decir, una que calcula la distancia de p a cada uno de los puntos de S y escoge el que proporciona distancia mínima) requiere tiempo $O(n)$, siendo n el número de puntos en S. Cuando n es grande y se ha de encontrar el vecino más próximo para muchos puntos, el tiempo necesario puede resultar prohibitivo.

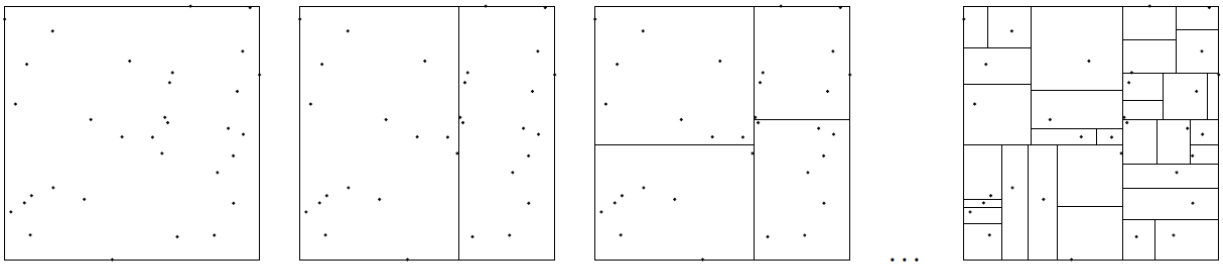
Hay muchos trabajos dedicados a este problema y un buen número de ellos trata con los denominados *kd-trees*. Un *kd-tree* es una estructura de datos que se calcula una sola vez sobre el conjunto S (y su construcción puede requerir tiempo sensiblemente superior a $O(n)$) y que facilita la búsqueda del vecino más próximo.

En este ejercicio nos centraremos exclusivamente en la construcción de los *kd-trees*. **No nos ocuparemos de cómo se utilizan una vez construidos.**

En la figura (a) puedes ver un conjunto S formado por 32 puntos en el plano ($m = 2$). Un *kd-tree* divide recursivamente los puntos en pares de conjuntos disjuntos. Lo hace escogiendo un eje (horizontal o vertical) y un valor sobre el eje escogido de acuerdo con un criterio de partición.

El criterio de partición "estándar" nos hace escoger el eje con mayor extensión: el que presenta mayor diferencia entre sus valores máximo y mínimo para los puntos de S (y uno cualquiera en caso de empate). En nuestro ejemplo, escogemos el eje horizontal (eje x). Como valor sobre dicho eje se escoge la mediana[1] de los valores de los puntos en dicho eje (las x de los puntos).

El procedimiento sigue construyendo, recursivamente, dos nuevos *kd-trees*: uno para los 16 puntos a la izquierda de la línea de partición y otro parta los 16 puntos a la derecha (véase la figura (b)).



La figura (c) muestra cómo sigue el proceso de selección/partición en los *kd-trees* izquierdo y derecho. El proceso recursivo finaliza cuando cada región contiene un solo punto. La figura (d) muestra el resultado final.

Para que podamos generar el *kd-tree* correctamente, el conjunto de puntos de partida no debe contener puntos repetidos. Recordad que la recursividad termina cuando queda un único punto en cada región y si hubiese dos puntos iguales no podríamos separarlos en dos regiones.

[1] El valor de la mediana de un conjunto de números depende de si el número de puntos es par o impar, consulta la Wikipedia para ver la forma correcta de calcular la mediana.

Como estructura de datos, un *kd-tree* es un árbol binario que representa las particiones efectuadas. Sus nodos internos contienen la siguiente información:

- El eje seleccionado: horizontal o vertical.
- El valor de partición, que es un número flotante (el valor que, en el eje elegido, parte el conjunto de puntos en dos conjuntos disjuntos).
- El hijo izquierdo, que es un *kd-tree* con el que se modelan los puntos a un lado del valor de partición en el eje seleccionado (en el caso de seleccionar el eje horizontal, representa a los puntos que hay a la izquierda del valor de partición y, en el otro caso, a los que se encuentran bajo dicho valor).
- El hijo derecho, que es otro *kd-tree* con el que se modelan los restantes puntos.

Cada nodo hoja contiene un punto de S.

Nosotros utilizaremos la siguiente implementación de *kd-trees* (disponible en `aux_e4.zip`):

```
class Axis:
    X = 0
    Y = 1

class KDTree(ABC):
    @abstractmethod
    def pretty(self, level: int = 0) -> str: pass

class KDNode(KDTree):
    def __init__(self, axis: Axis, split_value: float, child1: KDTree, child2: KDTree):
        self.axis = axis                # Eje utilizado para separar sus hijos (Axis.X o Axis.Y)
        self.split_value = split_value  # Coordenada x o y (depende de axis)
        self.child1 = child1            # Hijo izquierdo o superior (coordenada < split_value)
        self.child2 = child2            # Hijo dererecho o inferior (coordenada >= split_value)

    def pretty(self, level: int = 0) -> str:
        # Devuelve una cadena con la información del nodo y de sus descendientes.

class KDLeaf(KDTree):
    def __init__(self, point: Tuple[float, float]):
        self.point = point

    def pretty(self, level: int = 0) -> str:
        # Devuelve una cadena con la información de la hoja.
```

Para este entregable te pedimos:

a) Desarrollo e implementación (fichero **`entregable4.py`**):

- Implementa la función

  `def` **`read_points`**`(filename: str) -> List[Tuple[float, float]]`

  que reciba el nombre de un fichero con puntos 2D (dos números **flotantes** en cada línea, separados por un blanco) y devuelva una lista de tuplas `(x, y)`. Asumiremos que el fichero no contiene puntos repetidos.

- Implementa la función

  `def` **`build_kd_tree`**`(points: List[Tuple[float, float]]) -> KDTree`

  que reciba una lista de tuplas `(x, y)` y construya un árbol binario utilizando como nodos objetos de la clase `KDTree` (`KDLeaf` para las hojas y `KDNode` para el resto de nodos). La función debe devolver el nodo raíz de dicho árbol. La función debe seguir la estrategia de "divide y vencerás", aunque no se recomienda el esquema formal debido a que, en este caso, complica el código. Debes identificar los elementos del esquema "divide y vencerás" en tu código.

IMPORTANTE: Al elegir la línea de partición es posible que haya puntos justo en la línea. Para que todos obtengamos la misma solución deberemos asignarlos todos al hijo **child2**. Habrá al menos un punto sobre la línea cuando el numero de puntos de la partición sea impar (el punto del que obtenemos la mediana).

- Utilizando las dos funciones anteriores, **implementa el programa `entregable4.py`** que reciba como parámetro el nombre de un fichero de puntos, obtenga el *kd-tree* siguiendo el criterio de partición "estándar" presentado anteriormente y, por último, que lo imprima por pantalla. Dado que las clases `KDNode` y `KDLeaf` de `partida` tienen implementado el método `pretty()` que devuelve la información del objeto como una cadena, puedes hacerlo fácilmente con `print(kdtree.pretty())`.

b) Memoria del entregable. Una memoria en la que se detallen los pasos seguidos y dificultades experimentadas, a la vez que se presente una solución para el problema en cuestión. También debe contener el análisis del coste temporal y espacial de la construcción del *kd-tree* para n puntos. Las faltas de ortografía penalizan; una redacción descuidada penaliza.

c) Actas de las reuniones mantenidas hasta la entrega. Recordad que uno de vosotros será el secretario y se encargará de tomar nota en la/s reunión/es de trabajo. El cargo de secretario es rotativo: será una persona diferente para cada entregable.

d) Valoración personal. Cada miembro del grupo deberá escribir una breve valoración del trabajo realizado y de los resultados obtenidos.

Los apartados b, c y d se entregarán en papel, en clase, en la fecha indicada. Como respuesta a la tarea correspondiente en el aula virtual se enviará un fichero comprimido con todo (a, b, c y d).

Ficheros auxiliares. Pruebas públicas

En el aula virtual podéis descargaros el fichero `aux_e4.zip` con el siguiente contenido:

- `kdtree.py`: las clases que permiten representar kd-trees.
- `kdtreeviewer.py`: un visualizador, con EasyCanvas, de *kd-trees*.
- Un conjunto de pruebas públicas formado por cincos ficheros de puntos (con 3, 4, 10, 100 y 1000 puntos) junto con los cinco ficheros con los *kd-trees* correspondientes.

También podéis utilizar el programa `kdtreeviewer.py` para visualizar vuestros *kd-trees* con EasyCanvas (no olvidéis configurar PYTHONPATH):

```
> export PYTHONPATH=/.../algoritmia/src:/.../easycanvas/src/
> python3 entregable4.py points2d_1000.txt > kdtree_sol_1000.txt
> python3 kdtreeviewer.py kdtree_sol_1000.txt
```

Recuerda que en Windows la variable de entorno PYTHONPATH se define así (cambiando los puntos por la ruta correspondiente):

```
> set PYTHONPATH=c:\...\algoritmia\src;c:\...\easycanvas\src
```

Fecha de entrega de la memoria: lunes 10 de diciembre de 2018, en clase.

Fecha de entrega al aula virtual: viernes 7 de diciembre de 2018.

No os quedéis colgados. Recordad que hay tutorías.