UA

Universidad
de Alicante

ESCUELA POLITÉCNICA SUPERIOR

RAZONAMIENTO Y REPRESENTACIÓN DEL CONOCIMIENTO

Curso académico 2024 / 2025
Semestre 1
Grado en Ingeniería en Inteligencia Artificial
Grupo 1

Profesora: Diego Viejo Hernando
Alumno: Santiago Álvarez Geanta

ÍNDICE

Librerías utilizadas

En nuestro fichero **Ejercicio1.py** he utilizado la librería **numpy** para realizar todos los cálculos matemáticos necesarios con matrices.

Rejilla Ocupación

Principalmente, nuestra rejilla de ocupación tendrá una variable denominada **rejilla.** Esta variable será de tipo **diccionario** donde la **clave** hará referencia a una celda del espacio "x, y, z" y su valor será la **cantidad de puntos** dentro de dicha celda. Cabe añadir que nuestra clase recibe un parámetro denominado **tam_celda** que hace referencia al tamaño que cada celda tendrá en nuestra rejilla y otro parámetro que define los límites de nuestra rejilla.

Para obtener el valor de una celda en un espacio "x, y, z" simplemente realizaremos una división de la coordenada respecto al valor que tiene nuestra variable **tamaño de celda**. Habrá que tener en cuenta una posterior conversión a **entero** del valor obtenido al realizar la división.

```python
class RejjillaOcupacion:
    def __init__(self, tam_celda, limites):
        self.tam_celda = tam_celda
        self.limites = limites  # (xmin, xmax, ymin, ymax, zmin, zmax)
        self.rejilla = {}
        self.inicializarCeldas()

    def getCelda(self, x, y, z):
        # Índice de celda correspondiente a (x, y, z)
        return (
            int(x // self.tam_celda),
            int(y // self.tam_celda),
            int(z // self.tam_celda)
        )
```

Nuestro constructor también se encargará de inicializar cada celda de nuestra rejilla mediante una llamada al método inicializarCeldas. Nuestro método itera cada celda para posteriormente almacenarla en la variable rejilla, los valores almacenados serán valores por defecto. Nos apoyamos en el uso de **np.arange** para poder iterar usando valores **float**.

```python
def inicializarCeldas(self):
    # Genera todas las celdas posibles dentro de los límites
    xmin, xmax, ymin, ymax, zmin, zmax = self.limites
    # arange permite iterar usando float
    for x in np.arange(xmin, xmax, self.tam_celda):
        for y in np.arange(ymin, ymax, self.tam_celda):
            for z in np.arange(zmin, zmax, self.tam_celda):
                celda = self.getCelda(x, y, z)
                # Guardamos la suma en vez de la media de los puntos en esa celda
                # Para calcular la media -> suma / numero
                self.rejilla[celda] = {"numero": 0, "suma": np.zeros(3)}
```

El siguiente método que comenzaremos a desarrollar será uno de los más importantes y el encargado de almacenar los puntos de una muestra **.pcd**. Nuestro método comprobará si una determinada celda existe en nuestra rejilla. En caso afirmativo **incrementaremos el número de puntos** y guardaremos la suma de las coordenadas de este punto (no guardamos la media porque esta puede calcularse simplemente haciendo la operación suma / número).

```
def setPunto(self, x, y, z):
    celda = self.getCelda(x, y, z)

    if celda in self.rejilla:
        self.rejilla[celda]["numero"] += 1
        self.rejilla[celda]["suma"] += np.array([x, y, z])
```

Finalmente nos quedará diseñar el método para devolver los resultados del análisis. Este método iterará todas las celdas que hayamos almacenado en nuestra rejilla, por cada celda que no tenga puntos incrementaremos la variable **celdas_vacias;** en cambio, por cada celda que contenga puntos incrementaremos la variable **celdas_ocupadas**. Además, almacenaremos en una variable auxiliar la suma de los puntos que tiene cada celda para posteriormente calcular la media realizando una división entre la cantidad de **celdas ocupadas**.

```
def analisis(self):
    celdas_vacias = 0
    celdas_ocupadas = 0
    media_puntos = 0

    for datos in self.rejilla.values():
        if datos["numero"] == 0:
            celdas_vacias += 1
        else:
            celdas_ocupadas += 1
            media_puntos += datos["numero"]

    return {
        "num_celdas": len(self.rejilla),
        "celdas_vacias": celdas_vacias,
        "celdas_ocupadas": celdas_ocupadas,
        "media_puntos": media_puntos / celdas_ocupadas
    }
```

OcTree

Nuestra clase consta de diversas variables, la primera de ellas hace referencia a los límites del espacio otorgado *(xmin, xmax, ymin, ymax, zmin, zmax)*. La segunda será una variable que indicará la profundidad a la que se encuentra una determinada instancia, seguida de la tercera variable que nos indicará la profundidad máxima a la que se puede expandir nuestro **OcTree**. Las siguientes variables corresponden al almacenamiento de los puntos que se sitúen dentro de la instancia actual, y al la lista de instancias de OcTree (equivalencia a los hijos o expansiones).

```
class OcTree:
    def __init__(self, limites, profundidad=0, max_profundidad=10):
        self.limites = limites  # Límite del cubo: (xmin, xmax, ymin, ymax, zmin, zmax)
        self.profundidad = profundidad
        self.max_profundidad = max_profundidad
        self.puntos = [] # Lista de puntos almacenados en este nodo
        self.hijos = None  # Los 8 subnodos (hijos)
```

El primer método a desarrollar nos ayudará a expandir nuestro OcTree. Simplemente guardaremos una lista con 8 instancias distintas de nuestra clase OcTree con un nivel más de profundidad en nuestra **variable de hijos.**

```
    def expandir(self):
        """ Dividimos el nodo en 8 subnodos"""
        x_min, x_max, y_min, y_max, z_min, z_max = self.limites
        x_mid = (x_min + x_max) / 2
        y_mid = (y_min + y_max) / 2
        z_mid = (z_min + z_max) / 2

        # Creamos los 8 subnodos
        self.hijos = [
            OcTree((x_min, x_mid, y_min, y_mid, z_min, z_mid), self.profundidad + 1, self.max_profundidad),
            OcTree((x_mid, x_max, y_min, y_mid, z_min, z_mid), self.profundidad + 1, self.max_profundidad),
            OcTree((x_min, x_mid, y_mid, y_max, z_min, z_mid), self.profundidad + 1, self.max_profundidad),
            OcTree((x_mid, x_max, y_mid, y_max, z_min, z_mid), self.profundidad + 1, self.max_profundidad),
            OcTree((x_min, x_mid, y_min, y_mid, z_mid, z_max), self.profundidad + 1, self.max_profundidad),
            OcTree((x_mid, x_max, y_min, y_mid, z_mid, z_max), self.profundidad + 1, self.max_profundidad),
            OcTree((x_min, x_mid, y_mid, y_max, z_mid, z_max), self.profundidad + 1, self.max_profundidad),
            OcTree((x_mid, x_max, y_mid, y_max, z_mid, z_max), self.profundidad + 1, self.max_profundidad)
        ]
```

El siguiente método será el más importante a desarrollar, se trata del método encargado de añadir puntos a nuestra clase. La primera comprobación será ver si hemos llegado a la máxima profundidad o si estamos ante una instancia sin hijos. En ambos casos, añadiremos el punto a la variable que almacena los puntos. Solamente en el caso de que tengamos más de 8 puntos y no hayamos llegado a la última capa de profundidad expandimos nuestro OcTree. Una vez expandido iteramos todos los puntos y los transferiremos al nodo hijo que contenga dicho punto realizando **pt = pop()** y posteriormente **hijo.setPunto(pt)**. En caso de añadir un punto a un nodo que ya tenga hijos o un nodo de capa intermedia simplemente comprobaremos cuál de todos los hijos contiene dicho punto y se lo añadiremos.

```
    def setPunto(self, punto):
        if self.profundidad == self.max_profundidad or not self.hijos:
            self.puntos.append(punto)
            if len(self.puntos) > 8 and self.profundidad < self.max_profundidad:
                self.expandir()
                while self.puntos:
                    pt = self.puntos.pop()
                    for child in self.hijos:
                        if child.contiene(pt):
                            child.setPunto(pt)
                            break
        else:
            for child in self.hijos:
                if child.contiene(punto):
                    child.setPunto(punto)
                    break
```

Para comprobar si un nodo contiene un punto realizaremos una comparación de coordenadas respecto a los límites establecidos.

```
def contiene(self, punto) -> bool:
    x_min, x_max, y_min, y_max, z_min, z_max = self.limites
    x, y, z = punto
    return x_min <= x < x_max and y_min <= y < y_max and z_min <= z < z_max
```

Finalmente nos quedará de nuevo diseñar el método para devolver los resultados del análisis. Este método utiliza una sintaxis recursiva. Como caso base establecemos no tener hijos (última capa). El caso recursivo itera todos los hijos y va añadiendo el correspondiente número de celdas y de celdas ocupadas.

```
def analisis(self):
    if not self.hijos:
        return {
            "num_celdas": 1,
            "num_puntos": len(self.puntos)
        }
    else:
        analisis = {"num_celdas": 0, "num_puntos": 0}
        for child in self.hijos:
            child_analisis = child.analisis()
            analisis["num_celdas"] += child_analisis["num_celdas"]
            analisis["num_puntos"] += child_analisis["num_puntos"]
        return analisis
```

Tan solo nos quedará la carga de datos y puntos en nuestras clases. Para cargar los datos de cada muestra **pcd** realizaremos una lectura del archivo y comenzaremos a guardar en nuestra variable **puntos** a partir de la línea **DATA ascii.**

```
# Carga de Datos.pcd
def cargar_datos_pcd(filePath):
    puntos = []
    with open(filePath, 'r') as archivo:
        leer_datos = False
        for linea in archivo:
            if leer_datos:
                valores = linea.strip().split()
                if len(valores) >= 3:
                    x, y, z = valores[0], valores[1], valores[2]
                    puntos.append((x, y, z))
            elif linea.startswith("DATA ascii"):
                leer_datos = True
    return puntos
```

Pruebas

Al ejecutar nuestro script con distintos tamaños de celda podremos ver una diferencia significativa en el tiempo de ejecución. Esto se debe a que con un tamaño menor de celda nuestro script tendrá que generar un mayor número de celdas, por tanto el tamaño de la Talla aumenta e iterar es más costoso.

Comparemos varios resultados dependiendo del tamaño de celda. En todos los resultados utilicé la muestra de datos **scan000.pcd**.

```
------------ 6 ---------------
Tamaño Celda -> 0.9

Rejilla -> {'num_celdas': 20966550, 'celdas_vacias': 20938176, 'celdas_ocupadas': 28374, 'media_puntos': 28.454077676746316}

OcTree -> {'num_celdas': 91876, 'media_puntos': 8.787550611694023}

TiempoEjecución -> 31.71208143234253
```

```
------------ 6 ---------------
Tamaño Celda -> 1

Rejilla -> {'num_celdas': 15280056, 'celdas_vacias': 15256045, 'celdas_ocupadas': 24011, 'media_puntos': 33.6248386156345}

OcTree -> {'num_celdas': 91876, 'media_puntos': 8.787550611694023}

TiempoEjecución -> 25.637280464172363
```

```
------------ 6 ---------------
Tamaño Celda -> 2

Rejilla -> {'num_celdas': 1929300, 'celdas_vacias': 1921157, 'celdas_ocupadas': 8143, 'media_puntos': 99.14773425027632}

OcTree -> {'num_celdas': 91876, 'media_puntos': 8.787550611694023}

TiempoEjecución -> 10.185423851013184
```

```
------------ 5 ---------------
Tamaño Celda -> 5

Rejilla -> {'num_celdas': 528, 'celdas_vacias': 421, 'celdas_ocupadas': 107, 'media_puntos': 569.5046728971963}

OcTree -> {'num_celdas': 22086, 'media_puntos': 2.7807660961695193}

TiempoEjecución -> 0.9356415271759033
```

Gracias al resultado de las ejecuciones podemos afirmar una complejidad empírica de e^x, en el que a mayor tamaño de celda, menor tiempo de ejecución.