Facultad de Ingenierías
Departamento de Ingeniería en Sistemas

Estrategia 2-Proyecto

Análisis y Diseño de Algoritmos

Alejandro Gonzalez Flores

Luis Felipe Giraldo

Jose David González Villa

Manizales, Caldas

2024

- **Macroalgoritmo:**

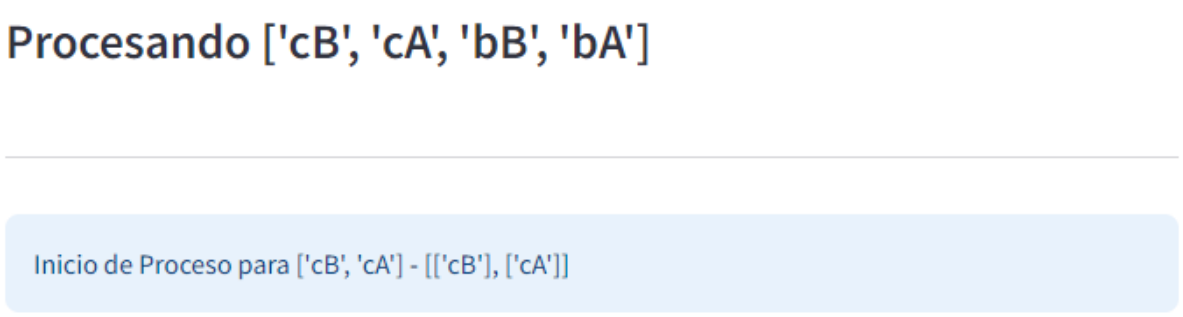
Comenzar con un conjunto A de todos los elementos. Es decir, A tendrá las aristas del subsistema definido a analizar.

a) Inicializar y , donde es un elemento arbitrario de A.

b) Iteración Principal

Para hasta (donde n es el número de aristas en) se calcula :

Encontrar que minimiza: donde es la función EMD entre la distribución de probabilidades resultante del subsistema sin las conexiones en y la distribución de probabilidades del subsistema completo, así:



Se ejecutarán todos los posibles candidatos y cada uno resultara con el resultado de los EMD Resultados

El calculo de EMD es 0.0

Final de Proceso para ['cB', 'cA']

c) Construcción de Pares:

El par forma un "par candidato" que tendrá una interpretación diferente al de la estrategia #1.

Inicio de Proceso para ['cB', 'cA', 'bB'] - [['cB', 'bB'], ['cA']]

Aquí ya se muestra las combinaciones que se han hecho

d) Recursión:

Si $|A| > 2$, repetir el proceso con $A' = A \mid \{a_{n-1}, a_n\} \cup \{u\}$, donde u representa la unión de a_{n-1} y a_n.
Continuar hasta que $|A| = 2$.

$$[['bB','bA'],['cB','cA']]$$

$$\bullet EMD : 0.0$$

En este momento se termina la ejecución que se hace la combinación de aristas que sean 2

e) Evaluación:

- Durante cada iteración en el proceso es necesario verificar si se genera una partición. En caso de que se genere una partición guardar su valor de pérdida y reemplazarla cuando se genere una partición con menor valor de pérdida.

Final de Proceso para ['cC', 'cB', 'bC']

$$\bullet NoTieneBiparticion$$

Para hacer esto creamos una clase llama graph que añadiremos los nodos y aristas:

```
class Graph:
    def __init__(self):
        self.nodes = []

    def add_node(self, value):
        """Agrega un nodo al grafo."""
        if not any(node.value == value for node in self.nodes):
            self.nodes.append(Node(value))

    def add_edge(self, value1, value2):
        """Crea una arista entre dos nodos."""
        node1 = next((node for node in self.nodes if node.value == value1), None)
        node2 = next((node for node in self.nodes if node.value == value2), None)

        if node1 and node2:
            if node2 not in node1.neighbors:
                node1.neighbors.append(node2)
            if node1 not in node2.neighbors:
                node2.neighbors.append(node1)
        else:
            raise ValueError("Ambos nodos deben existir en el grafo antes de agregar una arista.")

    def remove_edge(self, value1, value2):
```

Y ya con todos estos datos podremos saber cuanto es bipartido con la siguiente funcion:

```python
def has_bipartition(self):
    """Verifica si el grafo tiene una bipartición."""
    # Verifica primero si es conexo
    if not self.is_connected():
        return False  # Un grafo no conexo no puede tener bipartición válida

    color = {}

    for node in self.nodes:
        if node not in color:  # Nodo sin visitar
            queue = [node]
            color[node] = 0  # Asigna el primer color
            while queue:
                current = queue.pop(0)
                for neighbor in current.neighbors:
                    if neighbor not in color:
                        color[neighbor] = 1 - color[current]  # Colorea opuesto
                        queue.append(neighbor)
                    elif color[neighbor] == color[current]:
                        return False  # Dos nodos conectados tienen el mismo color
    return True
```

- Como al finalizar cada iteración se debe elegir la conexión que genera la menor diferencia, es decir se escoge cuál es el mínimo para todo , es posible que se presenten valores iguales, si esto sucede, escoger el que tenga el menor valor para .

La división con el menor valor de diferencia es la solución al problema

Eficiencia:

Parecida a la de la vez pasada ya que en la base es una estrategia parecida, la mayor diferencia radica en la cantidad de elementos, que en la primera se da solo por el número de nodos, mientras que en la segunda se define por el número de aristas que es igual al número de aristas que se determina por el número de nodos futuros y presentes.

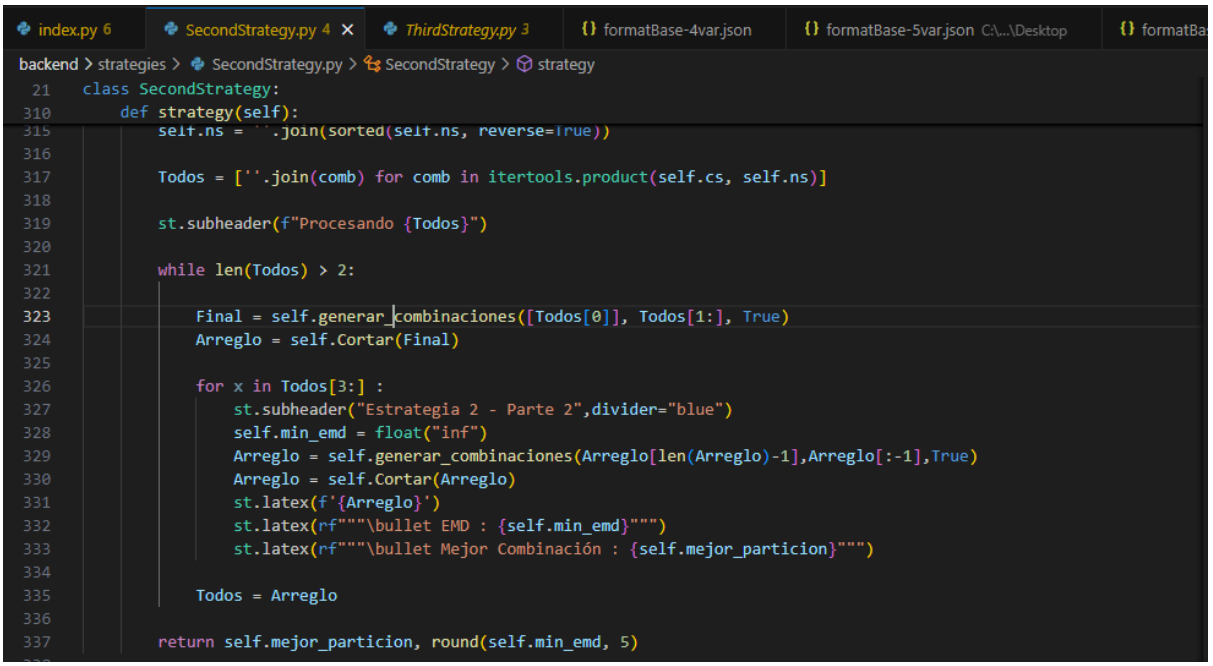
Esto hace que

$$N = (A*B)$$

Siento A los nodos futuros y los B los nodos presentes

Esto Implica que la Eficiencia es peor que la primera ya que esta hace el mismo proceso pero con mayor cantidad de valores por que las aristas que salen de los nodos en la gran mayoría

de los casos son mayores a los nodos, pero su fórmula de eficiencia es igual pero este “N” proviene es de las aristas



Como se puede observar las funciones que usa son las mismas.

$$= \sum_{i=1}^{N-2} \sum_{j=2}^{N-i} j$$

Para separarlas como sumatoria cuadrática podríamos separarlas para que queden

$$\frac{1}{2} \sum_{k=2}^{N-1} k^2 + \frac{1}{2} \sum_{k=2}^{N-1} k$$

Ahora resolvemos cada 1

$$\sum_{k=2}^{N-1} k^2 = \frac{(N-1)N(2N-1)}{6} - 1$$

$$\sum_{k=2}^{N-1} k = \frac{(N-1)N}{2} - 1.$$

Quedando la eficiencia:

$$\frac{1}{2}\left(\frac{(N-1)N(2N-1)}{6}-1\right)+\frac{1}{2}\left(\frac{(N-1)N}{2}-1\right)-(N-2)$$