Informaticae Abstracta

Año 2024, Vol. 2, No. 1 ENE | JUN 2024

Universidad Autónoma del Estado de México

Informaticae Abstracta
Año 2024, Vol. 2, No. 1 ENE | JUN 2024

Informaticae Abstracta, número 1, año 2023, es una revista de publicación continua editada por la Universidad Autónoma del Estado de México, con sede en el Instituto Literario número 100 oriente, Colonia Centro, Toluca, Estado de México, C.P. 50000. Se puede contactar a través del teléfono +(52 722) 2140855 y 21140795 ext. 10 1217, así como en el sitio web http://informaticae.uaemex.mx o mediante el correo electrónico: informaticae@uaemex.mx. La editora responsable es Ruth Hernández Pérez, perteneciente a la Facultad de Ingeniería. La revista cuenta con los derechos de uso exclusivo número 04-2022-111113523100-102 otorgados por el Instituto Nacional del Derecho de Autor. La última edición fue supervisada por Ruth Hernández Pérez, Javier Salas García y Mercedes Lucero Chávez, de la Facultad de Ingeniería, ubicada en Cerro de Coatepec s/n, Ciudad Universitaria, C.P. 50100, Toluca, Estado de México. La fecha de última modificación fue el 26/12/2023. Diseño de portada y diseño editorial: Ruth Hernández Pérez y Javier Salas García.

Informaticae Abstracta
Año 2024, Vol. 2, No. 1 ENE | JUN 2024

Índice

Informaticae Abstracta
Año 2024, Vol. 2, No. 1, pp. 4-24 ENE | JUN 2024

Algoritmos de clustering para la detección de posibles paradas de autobús

Sánchez Carmona, M. A.

Información del Artículo

Recibido: 3-may-2023
Aceptado: 18-ago-2023

Palabras clave: Clustering, k-means, DBSCAN, HDBSCAN, OPTICS, Códigos postales, Paradas de autobús, Haversine

Resumen

Este artículo propone una comparativa entre cuatro algoritmos de agrupamiento para la identificación de posibles paradas de autobús. Los algoritmos seleccionados fueron *k-means*, el *Agrupamiento Espacial Basado en Densidad de Aplicaciones con Ruido* (DBSCAN, por sus siglas en inglés), *Agrupamiento Espacial Jerárquico Basado en Densidad de Aplicaciones con Ruido* (HDBSCAN, por sus siglas en inglés) y el algoritmo de *Ordenación de Puntos para Identificar la Estructura de Clusters* (OPTICS, por sus siglas en inglés). El agrupamiento se aplicó a un conjunto de datos de una matrícula escolar utilizando la métrica de distancia *Haversine*. Los resultados del conteo de *clusters* y de *elementos no agrupados*, así como de la aplicación de métricas de calidad de los clusters (referidas en la literatura), mostraron que HDBSCAN tiene mejor desempeño que DBSCAN y OPTICS al generar una cantidad adecuada de clusters; por otro lado k-means mostró un mejor rendimiento computacional y mejores resultados de entre todos los algoritmos; sin embargo la cantidad de clusters son menores y por consiguiente inviables para ser usados como paradas de autobús.

1. Introducción

El agrupamiento (clustering, en inglés) consiste en formar grupos (clusters) con elementos que presentan características similares y se utiliza principalmente para determinar patrones climáticos, análisis cartográficos, análisis de datos, segmentación de imágenes, agrupación de artículos por temas o para segmentar clientes, entre otros [1]. Es una técnica de *Machine Learning*, que tiene como objetivo separar datos en grupos *homogéneos* con características comunes [2, 3]. Su uso en esta disciplina se enfoca en el *aprendizaje automático*, en la categoría del *aprendizaje no supervisado*; donde los dos problemas más comunes son el análisis de mixturas (mezclas) o agrupaciones y el aprendizaje de variables [3].

En el área del transporte el clustering ha sido utilizado para determinar y establecer paradas vehiculares, Liu y Wang [4] proponen un estudio enfocado en un "transporte en situación de emergencia", cuando el transporte ferroviario presenta fallas y los usuarios quedan varados; establecen un proceso de agrupamiento de datos en tiempo real, con el uso de un algoritmo de clustering DBSCAN mejorado y adaptativo, logrando determinar los puntos adecuados para establecer las paradas vehiculares donde se concentra la mayor cantidad de usuarios, con lo cual establecen rutas de transporte adecuadas para su modelo matemático.

Algunos algoritmos de clustering necesitan ser configurados para realizar el proceso de agrupamiento mediante un radio (distancia) de búsqueda para encontrar el elemento a añadirse al cluster. Sciortino et al. [5] describen las características

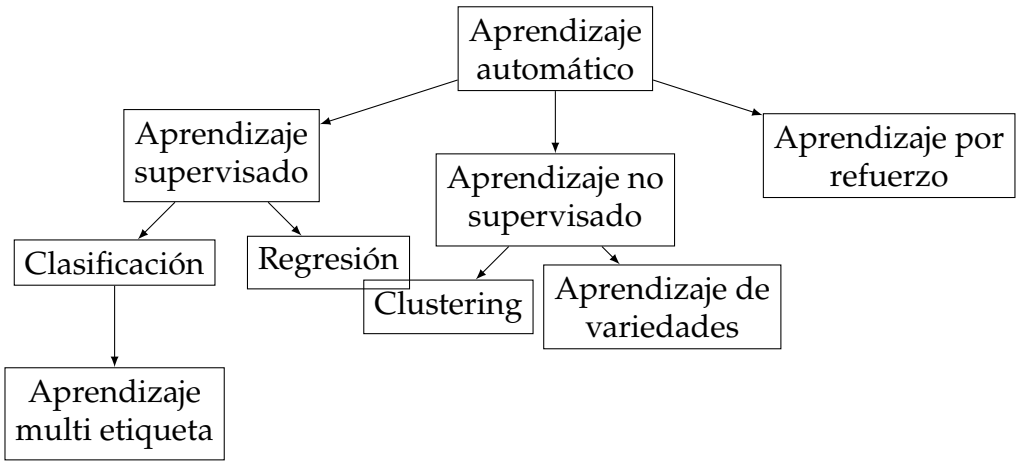


Figura 1: Esquema de las ramas principales del aprendizaje automático

de un Problema de Enrutamiento de Autobuses Escolares (SBRP, por su siglas en inglés), que es un problema del mundo real compuesto por varios subproblemas (selección de paradas de autobús, generación de rutas de autobús, entre otros), establecen mediante estudios estadísticos, que los alumnos que utilizan un servicio de transporte escolar puede recorrer una distancia de 1,6 km para trasladarse a la parada de autobús. Con este tipo de estudios es factible determinar parámetros para los algoritmos de clustering que requieren ser configurados con una distancia.

El objetivo de este trabajo es comparar cuatro algoritmos de clustering para la generación de grupos con datos espaciales, utilizando una métrica de distancia adecuada que permita relacionar cada elemento o elementos para determinar si pertenecen o no a un grupo en particular y finalmente establecer puntos de parada para un servicio de autobuses. La métrica más utilizada es la distancia euclidiana, pero existen otras métricas como la distancia Manhattan, la distancia sobre una esfera con fórmula Haversine, entre otras [6]. Los algoritmos de clustering propuestos, bastante conocidos en el área del aprendizaje no supervisado, son k-means y DBSCAN, así como variaciones de este último como son HDBSCAN y OPTICS. En la siguientes secciones se describen los algoritmos de clustering utilizados, métricas de distancia y valoración de la calidad de los clusters, así como del preprocesamiento de datos realizados y los resultados obtenidos.

2. Aprendizaje no supervisado

En el área de la inteligencia artificial (IA) es común mencionar el término aprendizaje automático, considerado como una rama de la IA, la cual trata temas relacionados a la predicción y a la toma automática de decisiones, así como técnicas de representación, descripción y exploración de datos [3]. El aprendizaje automático cuenta con varias ramas, una clasificación breve se puede apreciar en la Figura 1. En la rama del *aprendizaje no supervisado* se encuentra la categoría de *clustering*, en dicha área radica una variedad de algoritmos cuya tarea es formar grupos de datos [1].

k-means

Es uno de los algoritmos de clustering más populares y ampliamente utilizados en el campo del aprendizaje automático no supervisado [6]. La letra k en k-means se refiere al número de clusters que el algoritmo intenta encontrar en un conjunto de datos. Su objetivo principal es particionar un conjunto de datos en k clusters, donde cada punto de datos pertenece al cluster cuyo *centroide* está más cerca [7].

Características y funcionamiento

- Se puede utilizar para segmentar clientes en diferentes grupos basados en sus características, como comportamientos de compra o preferencias.
- En minería de textos se puede agrupar documentos similares para tareas como la clasificación de textos o la recomendación de contenido.
- Puede utilizarse para reducir el número de colores en una imagen, lo que resulta en una representación comprimida de la misma.

Algoritmo

El algoritmo k-means opera de la siguiente manera:

1. Se comienza seleccionando aleatoriamente k puntos como centroides iniciales.
2. Cada punto de datos se asigna al cluster cuyo centroide está más cerca según alguna medida de distancia (comúnmente la distancia euclidiana).
3. Una vez que todos los puntos han sido asignados a un cluster, los centroides se recalculan como el promedio de todos los puntos asignados a ese cluster.
4. Los pasos 2 y 3 se repiten hasta que los centroides no cambien significativamente entre iteraciones o hasta que se alcance un número máximo de iteraciones.
5. El algoritmo converge cuando los centroides no cambian significativamente entre iteraciones o cuando se alcanza un número máximo de iteraciones predefinido.

El pseudocódigo del algoritmo k-means se muestra en el Algoritmo 1

Algorithm 1: Algoritmo k-means

kmeansdatos, k, max_iter centros ← inicializar_centros(datos, k) iteraciones ← 0 **while** iteraciones < max_iter **do**-_

clusters ← asignar_datos_a_clusters(datos, centros)
centros_nuevos ← calcular_nuevos_centros(datos, clusters) **if** centros_nuevos ≈ centros
_**then**

break centros ← centros_nuevos iteraciones ← iteraciones + 1 **return** clusters

La complejidad computacional promedio del algoritmo k-means esta dada por $O(k \cdot n \cdot T)$, donde k es el número de clusters, n es el número de muestras y T el

número de iteraciones; para el peor de los casos la complejidad del algoritmo es de $O(n^{\frac{k+2}{p}})$ con p siendo el número de características [8].

Agrupamiento Espacial Basado en Densidad de Aplicaciones con Ruido

Este algoritmo de agrupamiento de datos, conocido simplemente como DBSCAN, propuesto por Martin Ester et al. [9], se le denomina algoritmo de agrupamiento basado en densidad, porque encuentra un número de clusters comenzando por una estimación de la distribución de densidad de los nodos correspondientes. Es uno de los algoritmos de agrupamiento más usados y citados en la literatura científica [9].

Características y funcionamiento

DBSCAN presentan varias elementos que le permiten su eficiencia en el agrupamiento de datos espaciales, las características principales son:

- Para cada observación se mira el número de puntos a una distancia máxima eps de ella (esta zona se denomina eps-vecindad de la observación).
- Si una observación tiene al menos un cierto número de vecinos, incluida ella misma, se considera una observación central.
- Todas las observaciones en la vecindad de una observación central pertenecen al mismo grupo.
- Cualquier observación que no sea una observación central y que no tenga ninguna observación central en su vecindad se considera una *anomalía*.

Algoritmo

El proceso general que realiza DBSCAN es el siguiente:

1. Encontrar los puntos en la eps-vecindad de cada punto e identificar los puntos centrales con la mayor cantidad de $minPts$ vecinos.
2. Encontrar los componentes conectados de los puntos centrales en el gráfico vecino, ignorando todos los puntos no centrales.
3. Asignar cada punto no central a un grupo cercano si el grupo es un vecino eps; de lo contrario, asignarlo como un dato no etiquetado comúnmente denominado ruido.

Este proceso continúa hasta construir completamente un grupo densamente conectado. Entonces, un nuevo punto no visitado se visita y procesa con el objetivo de descubrir otro grupo o ruido.

El algoritmo original DBSCAN, retomado de [9], se compone de tres elementos: algoritmo principal 2, la función *ExpandCluster* 3 y el método *regionQuery* 4.

Algorithm 2: DBSCAN

```
setOfPoints, eps, minPts n ← size(setOfPoints) clusterId ← UNCLASSIFIED
visited[n] ← FALSE clusterAssignment ← nelementos, inicializadosa0 for i ← 1 TO n
do-_
visited[i] ← TRUE continue visited[i] = TRUE
_neighbors = regionQuery(setOfPoints, i, eps) if size(neighbors) < minPts then
_clusterAssignment[i] ← NOISE else
clusterId ← clusterId + 1 ExpandCluster(setOfPoints, i, neighbors, clusterId, eps, minPts,
            visited, clusterAssignment)
return clusterAssignment
```

Algorithm 3: ExpandCluster

```
setOfPoints, point, neighbors, clusterId, eps, minPts, visited,
            clusterAssignment clusterAssignment[point] ← clusterId i ← 1 while
i ← size(neighbors) do-_
_p ← neighbors[i] if !visited[p] then
visited[p] ← TRUE newNeighbors ← regionQuery(setOfPoionts, p, eps) if
_size(newNeighbors) ⩾ minPts then
_append(neighbors, newNeighbors) if clusterAssignment[p] == 0 then
clusterAssignment[p] ← clusterId i = i + 1
```

Algorithm 4: RegionQuery

```
setOfPoints, eps neighbors[] n ← size(setOfPoints) for j ← 1 TO n do-_
lat1 ← setOfPoints[i][1] lon1 ← setOfPoints[i][2] lat2 ← setOfPoints[j][1]
_lon2 ← setOfPoints[j][2] if distanceHaversine(lat1, lon1, lat2, lon2) ⩽ eps then
neighbors.push(j)
return neighbors
```

Complejidad

DBSCAN visita cada punto de la base de datos, posiblemente varias veces (e.g., como candidatos a diferentes clusters). Para la práctica, sin embargo, la complejidad temporal es mayormente gobernada por el número de llamados a regionQuery. DBSCAN ejecuta exactamente una consulta por cada punto, y si se utiliza una estructura de índice que ejecute la consulta a los vecinos (neighborhood query) en $O(\log n)$, la complejidad temporal total sería $O(n \log n)$. Sin usar la estructura de índice, la complejidad temporal sería $O(n^2)$. A menudo, la matriz de distancias de tamaño $(n^2 - n)/2$ se construye para evitar recalcular las distancias. Esto necesita $O(n^2)$ de memoria, mientras que con una implementación que no se base en matrices solo se necesita $O(n)$ de memoria [9, 10].

Agrupamiento Espacial Jerárquico Basado en Densidad de Aplicaciones con Ruido

Conocido como HDBSCAN, es un algoritmo de clustering avanzado que se basa en densidad y jerarquía, es una extensión del algoritmo DBSCAN, debido a que éste puede tener dificultades para capturar con éxito grupos con diferentes densidades, mientras que HDBSCAN explora todas las escalas de densidad posibles mediante

la construcción de una representación alternativa del problema de agrupación [11]. Se utiliza principalmente para identificar patrones y estructuras en grandes conjuntos de datos, especialmente cuando la forma y la densidad de los clusters no son uniformes [11, 12].

Características principales y funcionamiento

- Agrupa puntos de datos que están densamente conectados, al igual que DBSCAN, lo que significa que hay un número suficiente de puntos en su vecindad.
- Crea una jerarquía de clusters a diferencia de DBSCAN, esta jerarquía permite al algoritmo identificar clusters a diferentes niveles de granularidad.
- Etiqueta los puntos que no pertenecen a ningún cluster como dato no etiquetado, lo que ayuda a identificar outliers (valores atípicos) en los datos.
- Es más robusto en comparación con DBSCAN en términos de la selección de parámetros. En lugar de depender de un solo valor de densidad mínima, construye una jerarquía y selecciona clusters basados en la estabilidad de estos.

Funcionamiento del Algoritmo

1. Transforma el espacio de datos en función de su densidad. Esto ayuda a identificar áreas con mayores concentraciones de puntos de datos.
2. Se crea un gráfico que conecta puntos de datos dentro de un cierto umbral de distancia, esto forma la base para identificar grupos.
3. Se identifican los componentes conectados en la gráfico, formando una estructura jerárquica de clusters potenciales.
4. Según el parámetro de tamaño mínimo del cluster, los clusters más pequeños se fusionan en otros más grandes y estables.
5. La solución de agrupación final se extrae de la jerarquía condensada, lo que garantiza que los grupos identificados sean resistentes a las variaciones de parámetros.

La complejidad algorítmica total de HDBSCAN es de aproximadamente O(N log N), lo que lo hace más eficiente que otros algoritmos jerárquicos que pueden tener complejidades cuadráticas o peores, especialmente para conjuntos de datos grandes [11, 12].

Ordenación de Puntos para Identificar la Estructura de Clusters

El algoritmo OPTICS es una técnica avanzada de clustering basada en densidad que se utiliza para descubrir estructuras de clusters en conjuntos de datos complejos. Es una extensión del algoritmo DBSCAN y aborda algunas de sus limitaciones, particularmente en la identificación de clusters con densidades variables [13]. Se utiliza en una amplia gama de aplicaciones similares a DBSCAN, incluyendo:

- Análisis de datos geoespaciales: Identificación de regiones con alta densidad de eventos geográficos.
- Bioinformática: Agrupación de genes o proteínas con características similares.
- Análisis de mercado: Segmentación de clientes basado en comportamientos de compra.
- Procesamiento de imágenes: Agrupación de características de imágenes.

Características principales y funcionamiento

- Agrupa puntos de datos que están densamente conectados al igual que DBSCAN.
- No produce directamente una asignación de clusters, en su lugar genera una ordenación lineal de puntos de datos que representa la estructura de densidad subyacente.
- La salida principal es un gráfico de "reachability", que visualiza las distancias de accesibilidad de los puntos en el orden procesado, ayudando a identificar clusters de diferentes densidades.
- Maneja mejor los clusters con diferentes densidades en comparación con DBSCAN.

Los parámetros principales que utiliza OPTICS son:

- $min\_samples$: Número mínimo de puntos requeridos para que un punto sea considerado núcleo.
- epsilon (ϵ): Radio máximo de búsqueda de vecinos. En OPTICS, este parámetro es menos crítico ya que el algoritmo explora un rango de valores de ϵ.

Funcionamiento del Algoritmo

1. Se calcula para cada punto una "distancia de accesibilidad" que indica la distancia mínima necesaria para alcanzar este punto desde un punto núcleo.
2. Se ordenan los puntos en función de sus distancias de accesibilidad, generando una representación ordenada de la estructura de densidad del conjunto de datos.
3. A partir de la ordenación de distancias de accesibilidad, se pueden extraer clusters utilizando diferentes técnicas de post-procesamiento, como la identificación de valles en el gráfico de reachability.

La complejidad algorítmica total de OPTICS, en el peor de los casos, es de $O(N^2)$, cuando el algoritmo debe considerar todos los puntos como vecinos en un conjunto de datos denso [13, 14].

3. Comparativa de los algoritmos de clustering

En el Cuadro 1 se resumen las principales ventajas y desventajas de cada uno de los algoritmos revisados en este trabajo.

Tabla 1: Ventajas y desventajas de k-means, DBSCAN, HDBSCAN y OPTICS

Algoritmo	**Ventajas**	**Desventajas**
k-means	Es fácil de comprender y de implementar. Es relativamente rápido y eficiente en términos de tiempo de ejecución. Converge rápidamente a una solución, lo que lo hace adecuado para aplicaciones en tiempo real. Puede manejar grandes volúmenes de datos y puede ser fácilmente paralelizado.	Requiere que el número de clusters k sea especificado previamente. Los resultados del algoritmo pueden variar significativamente dependiendo de la inicialización de los centroides. Asume que los clusters son esféricos y de tamaño similar, lo cual puede no ser apropiado para datos con clusters de formas arbitrarias o densidades variadas.
DBSCAN	No requiere que se especifique el número de clusters en los datos. Posee una noción de ruido y es resistente a los valores atípicos. Requiere solo dos parámetros y es prácticamente insensible al orden de los puntos en los datos usados. Los parámetros `minPt` y *eps* pueden ser establecidos por un experto en el dominio, si los datos se comprenden bien.	Los puntos fronterizos a los que se puede llegar desde más de un grupo pueden formar parte de cualquiera de ellos (dependiendo del orden en que se procesen los datos). La calidad depende de la métrica de distancia utilizada. No puede agrupar bien conjuntos de datos con grandes diferencias en densidades.

Algoritmo	Ventajas	Desventajas
HDBSCAN	Puede identificar clusters de formas arbitrarias. Es eficaz en la identificación de puntos de ruido, lo cual mejora la calidad del clustering al excluir datos ruidosos que podrían distorsionar los resultados. No se necesita especificar el número de clusters. Produce una jerarquía de clusters, proporcionando una visión más detallada de la estructura de los datos a diferentes niveles de granularidad.	Requiere la selección adecuada de parámetros como $min\_samples$ (minPts) y $min\_cluster\_size$, lo que puede ser complejo sin conocimiento previo de los datos. Aunque es escalable, para conjuntos de datos extremadamente grandes puede requerir un tiempo de computación significativo y recursos de memoria elevados. El rendimiento y los resultados del algoritmo dependen en gran medida de la métrica de distancia utilizada.
OPTICS	No requiere que se especifique el número de grupos en los datos. Puede manejar conjuntos de datos con clusters que tienen diferentes densidades. El gráfico de reachability proporciona una visualización intuitiva de la estructura de densidad, facilitando la identificación de clusters y outliers.	Es más complejo computacionalmente en comparación con DBSCAN debido a la necesidad de calcular y ordenar las distancias de accesibilidad. La salida requiere pasos adicionales de post-procesamiento para la extracción de clusters, lo que puede ser menos intuitivo.

Métricas para la determinación de la distancia

Para los algoritmos de clustering basados en densidad, el definir una métrica de distancia es un factor determinante para la cantidad de clusters que se pueden formar [6], en DBSCAN es más notorio este factor debido al parámetro eps, por otro lado HDBSCAN y OPTICS tienen la particularidad de establecer dicho parámetro mediante cálculos [11, 13]. La distancia puede ser dada bajo una métrica en particular dependiendo del contexto de los datos, es decir puede ser usada una distancia euclidiana, Haversine, entre otras. Las principales métricas de distancia utilizadas son:

Distancia Euclidiana

Es un número positivo que indica la separación de dos puntos en un espacio donde se cumplen los axiomas y teoremas de la geometría de Euclides [6]. La distancia entre dos puntos $p = (x_1, y_1)$ y $q = (x_2, y_2)$ de un espacio euclidiano es la longitud del vector pq perteneciente a la única recta que pasa por dichos puntos. Para un espacio de dos dimensiones, la fórmula de la distancia euclidiana se muestra en la Ecuación 1.

$$d(p, q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1}$$

En un espacio n-dimensional, la distancia Euclideana entre los puntos $P = (p_1, p_2, \ldots, p_n)$ y $Q = (q_1, q_2, \ldots, q_n)$ se puede definir en la Ecuación 2.

$$d_E(P, Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)} \tag{2}$$

Distancia Manhattan

También conocida como distancia de la ciudad o distancia L_1, es una métrica de distancia utilizada en el análisis de datos y aprendizaje automático. Se llama así porque representa la distancia que recorrería un taxi en una cuadrícula de calles rectangulares, similar al sistema de calles de Manhattan [6]. La distancia Manhattan entre dos puntos $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$ en un espacio bidimensional se calcula como:

$$Distancia\ Manhattan = |x_1 - x_2| + |y_1 - y_2| \tag{3}$$

Si se trata de un espacio de más dimensiones entre dos puntos $p = (p_1, p_2, \ldots, p_n)$ y $q = (q_1, q_2, \ldots, q_n)$ en un espacio de n-dimensional.

$$Distancia\ Manhattan = \sum_{i=1}^{n} |p_i - q_i| \tag{4}$$

Fórmula Haversine

Es una fórmula utilizada para calcular la distancia entre dos puntos de una esfera dadas sus coordenadas de longitud y latitud. En términos más sencillos, calcula la distancia más corta entre dos puntos de la superficie de un objeto esférico, como la Tierra. La fórmula de Haversine, ver Ecuación 5, se utiliza para calcular rutas de vuelo, distancias en mapas y navegación marítima [15]. También se utiliza en meteorología para calcular las trayectorias de las tormentas, para posicionar equipos de telecomunicaciones y para trazar la trayectoria de los satélites en el cielo [16, 17].

$$d = 2 \cdot R \cdot \arcsin \sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)} \tag{5}$$

donde:

- φ_1, φ_2 son las latitudes del punto 1 y 2 respectivamente.

- λ_1, λ_2 son las longitudes del punto 1 y 2 respectivamente.
- R es el radio de la esfera (en este caso el radio de la Tierra, que equivale a $6,371$ km).

La fórmula Haversine no puede ser usada por más de dos dimensiones, por lo que en el contexto de clustering no permite realizar un análisis de diversas características numéricas a la vez, en esas situaciones es ideal utilizar distancia Euclideana o Manhattan.

Métricas de evaluación para algoritmos de clustering

Para determinar qué algoritmos de clustering son mejores para un conjunto de datos, es necesario el uso de métricas de evaluación. Algunas de las métricas más comunes son:

Silhouette Score

Mide cuán similares son los puntos dentro de un cluster en comparación con los puntos de otros clusters. Su valor oscila entre -1 y 1, donde 1 indica clusters bien definidos [18]. El proceso del cálculo de Silhouette Score es el siguiente:

1. Se calcula la distancia promedio $a(i)$ entre el punto i y todos los demás puntos en el mismo clúster mediante la Ecuacion 6, siendo C_i el conjunto de puntos y $\|x_i - x_j\|$ la distancia entre los puntos i y j.

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, j \neq i} \|x_i - x_j\| \tag{6}$$

2. Se calcula de la distancia promedio $b(i)$ entre el punto i y todos los puntos en el cluster más cercano (que no sea el propio cluster de i) mediante la Ecuación 7, donde C_k es un cluster diferente al cluster C_i.

$$b(i) = \min_{k \neq C_i} \frac{1}{|C_k|} \sum_{j \in C_k} \|x_i - x_j\| \tag{7}$$

3. Se realiza el cálculo del índice de Silhouette para el punto i con la Ecuación 8.

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{8}$$

4. Finalmente se calcula el Silhouette score para el conjunto completo con la Ecuación 9, siendo esta el promedio del índice de Silhouette de todos los puntos, donde N es el número total de puntos en el conjunto de datos.

$$S = \frac{1}{N} \sum_{i=1}^{N} s(i) \tag{9}$$

Índice Davies-Bouldin

Evalúa la dispersión de los clusters y la distancia entre ellos. Un valor más bajo indica clusters mejor formados [19]. El procedimiento para obtener el índice Davies-Bouldin es el siguiente:

1. Se obtienen la dispersión de cada cluster utilizando la Ecuación 10, siendo C_i es el conjunto de puntos, c_i es el centroide del cluster, $\|x - c_i\|$ la distancia entre el punto x y el centroide, y $|C_i|$ es el número de puntos en el cluster todo esto para el punto i.

$$S_i = \frac{1}{|C_i|} \sum_{x \in C_i} \|x - c_i\| \tag{10}$$

2. El cálculo de la separación entre clusters se consigue aplicando la Ecuación 11, donde c_i y c_j son los centroides de los clusters i y j respectivamente y $\|c_i - c_j\|$ siendo la distancia entre los centroides de los clusters.

$$M_{ij} = \|c_i - c_j\| \tag{11}$$

3. La medida de Davies-Bouldin para cada par de clusters se obtienen con la Ecuación 12.

$$R_{ij} = \frac{S_i + S_j}{M_{ij}} \tag{12}$$

4. Se calcula el máximo de R_{ij} para cluster i sobre todos los clusters $j \neq i$ utilizando la Ecuación 13.

$$D_i = \max_{j \neq i} R_{ij} \tag{13}$$

5. Finalmente se obtienen el índice de Davies-Bouldin con la Ecuación 14, donde N es el número total de clusters.

$$DBI = \frac{1}{N} \sum_{i=1}^{N} R_i \tag{14}$$

Índice Calinski-Harabasz

También conocido como el criterio de varianza de la relación, evalúa la dispersión entre clusters y la cohesión dentro de los clusters. Un valor más alto indica clústeres mejor formados [20].

1. Se obtienen la matriz de dispersión entre clústeres (B_k) mediante la Ecuación 15, siendo k el número de clusters, C_i es el conjunto de puntos en el cluster i, $|C_i|$ es el número de puntos en el cluster i, c_i es el centroide del cluster iy c es el centroide global de todos los datos.

$$B_k = \sum_{i=1}^{k} |C_i|(c_i - c)(c_i - c)^T \tag{15}$$

2. Se calcula la matriz de dispersión dentro de clusters W_k con la Ecuación 16

$$W_k = \sum_{i=1}^{k} \sum_{x \in C_i} (x - c_i)(x - c_i)^T \tag{16}$$

3. El cálculo del índice de Calinski-Harabasz (CH), se obtienen finalmente aplicando la Ecuación 17, donde $trace(B_k)$ es la traza de la matriz de dispersión entre clusters, $trace(W_k)$ es la traza de la matriz de dispersión dentro de clusters y n es el número total de puntos en el conjunto de datos.

$$CH = \frac{trace(B_k)/(k-1)}{trace(W_k)/(n-k)} \tag{17}$$

4. Procesamiento de los datos

Los datos utilizados en los algoritmos de clustering, se conforman de la matrícula de alumnos, compuesto por 96,604 registros, cada registro contienen la siguiente información: Nombre del espacio académico (EA), calle, población, estado, municipio, código postal (CP). Después de una limpieza a los datos, el conteo de registros pasó a ser de 96,113. Para poder realizar el proceso de clustering con coordenadas geográficas, se tomó el CP para realizar una identificación aproximada de la zona donde cada alumno radica, en términos de longitud y latitud; para ello se realizó el siguiente proceso:

1. Identificar el rango de códigos postales para establecer la zona de servicio de transporte, donde se requieren identificar posibles paradas de autobús, para este trabajo se tomaron los códigos postales en el rango de 50000 a 52077 de México.

2. Cada código postal, en el rango descrito, debe ser mapeado a una coordenada geográfica en formato (longitud, latitud), por lo cual se utilizó la API (Application Programming Interface) que proporciona el servicio de OpenStreetMap [21], el cual incluye un método de consulta sobre la información de códigos postales en un determinado país, su uso para el presente trabajo es la conversión de códigos postales a coordenadas geográficas, mediante un solicitud *GET*. Con este proceso, se construyó un archivo con la asignación CP a Longitud, Latitud.

3. Al realizar un filtrado aplicado a los datos de la matrícula de alumnos y obtener aquellos registros con el rango de CP, determinado anteriormente, se obtuvo un total de 41,298 registros para realizar el proceso de clustering.

4. Debido a que la matrícula de alumnos contenía registros con CP iguales, se realizó un proceso para obtener los CP "únicos", es decir, CP donde radican los alumnos sin repetición de los mismos, este proceso permite eficientar los algoritmos de clustering reduciendo la cantidad de datos a analizar para cada algoritmo.

Los estructura de los datos obtenidos después del procesamiento se muestran en el Cuadro 2, estos datos son los utilizados para realizar el clustering con cada uno de los algoritmos.

Tabla 2: Muestra de los datos finales para el proceso de clustering

CP	Cantidad de alumnos	Longitud	Latitud
50000	93	-99.657693	19.291105
50010	1432	-99.646928	19.315436
50016	156	-99.605140	19.314034
50017	115	-99.614347	19.308198
...	...	...	...

5. Implementación de los algoritmos de clustering

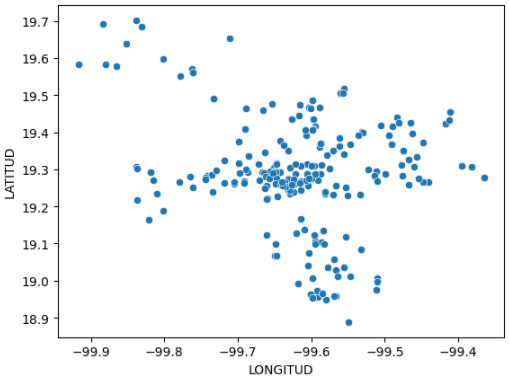
Dentro de los módulos que contienen la biblioteca *scikit-learn* se encuentra la definición de algunos algoritmos de agrupamiento para el lenguaje python, en los que se encuentran: *k-means, K-medoids, DBSCAN, HDBSCAN, OPTICS*, entre otros; además de métodos para evaluarlos [22].

Para el uso de los algoritmos de clustering en el lenguaje python con scikit-learn, se deben establecer algunos parámetros de configuración; en el caso particular de k-means se debe especificar la cantidad de clusters a formar mediante el parámetro $n\_clusters$. Por otro lado, en DBSCAN se deben especificar dos parámetros necesarios para el funcionamiento del algoritmo, estos son eps y $min\_samples$, el primero para establecer el radio de distancia, probado con valores de 1 km a 3 km, y el segundo para definir la cantidad de puntos mínimos, dejándolo en un valor fijo de 2 (dos puntos mínimos para se considerado cluster). En cuanto a HDBSCAN y OPTICS soló se estableció $min\_samples = 2$. Para los tres algoritmos de clustering basados en densidad se estableció la métrica Haversine.

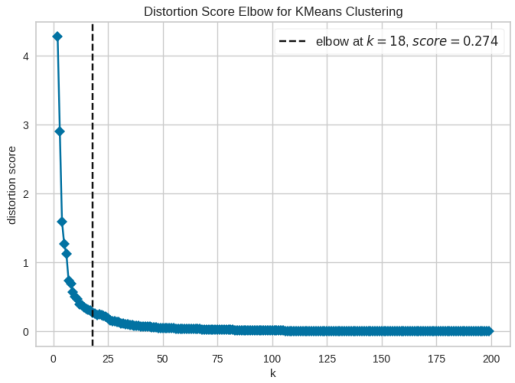
6. Resultados

Con el preprocesamiento descrito previamente, la gráfica de los datos de los códigos postales únicos con sus respectivas coordenadas se muestran en la Figura 2a. Mediante el uso de un gráfico de codo, ver Figura 2b, se pudo determinar el valor adecuado de clusters para configurar k-means, con una valor $k = 18$. Los resultados del agrupamiento por k-means se puede ver en la Figura 2c. Tomando estos resultados podemos determinar un máximo de 18 posibles paradas de autobús para el ámbito de transporte.

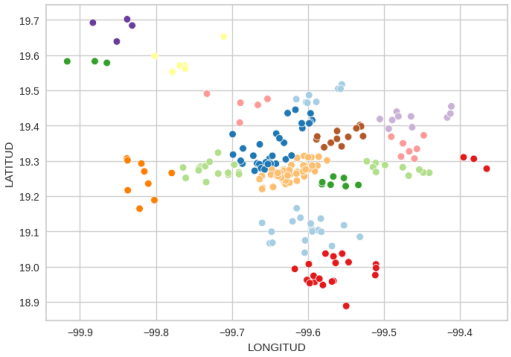
Los resultados obtenidos con la implementación de DBSCAN se muestran en la Tabla 3, donde se hizo variaciones al parámetro eps, con valores de 1,0 km, 1,5 km, 2,0 km, 2,5 km y 3,0 km; mientras que el parámetro $minPTS$ se mantuvo constante con un valor de 2, esto con el objetivo de contabilizar la cantidad de grupos que se generan, así como la cantidad de datos no agrupados, en la Figura 3 se puede apreciar los cambios en el conteo de clusters y datos no agrupados. Para la comparativa entre los otros algoritmos de densidad se tomó DBSCAN con los resultados obtenidos de $eps = 1{,}5$ km.



(a) Códigos postales en un plano cartesiano



(b) Gráfica de codo que muestra el mejor valor de k = 18



(c) 18 clusters formados con k-means

Figura 2: Proceso de clustering con k-means

Tabla 3: Resultados de las pruebas de agrupamiento mediante DBSCAN

Resultados del clustering con DBSCAN			
Eps	**minPts**	**Cantidad de clusters**	**Datos no etiquetados**
1	2	29	126
1.5	2	28	84
2	2	32	54
2.5	2	26	30
3	2	21	22

El proceso de clustering con DBSCAN, HDBSCAN y OPTICS se muestran en las Figuras 4a, 4b y 4c respectivamente, se puede observar que OPTICS tiende a formar una mayor cantidad de grupos en comparación de los otros algoritmos basados en densidad, lo cual es un resultado contrario a lo que determina k-means. Para el área de transporte de pasajeros, es un punto importante ya que se tienen mayor cantidad de paradas, pero adecuadas ya que no son de un área muy extensa y por ende los pasajeros pueden trasladarse de sus residencias al punto de parada determinado.

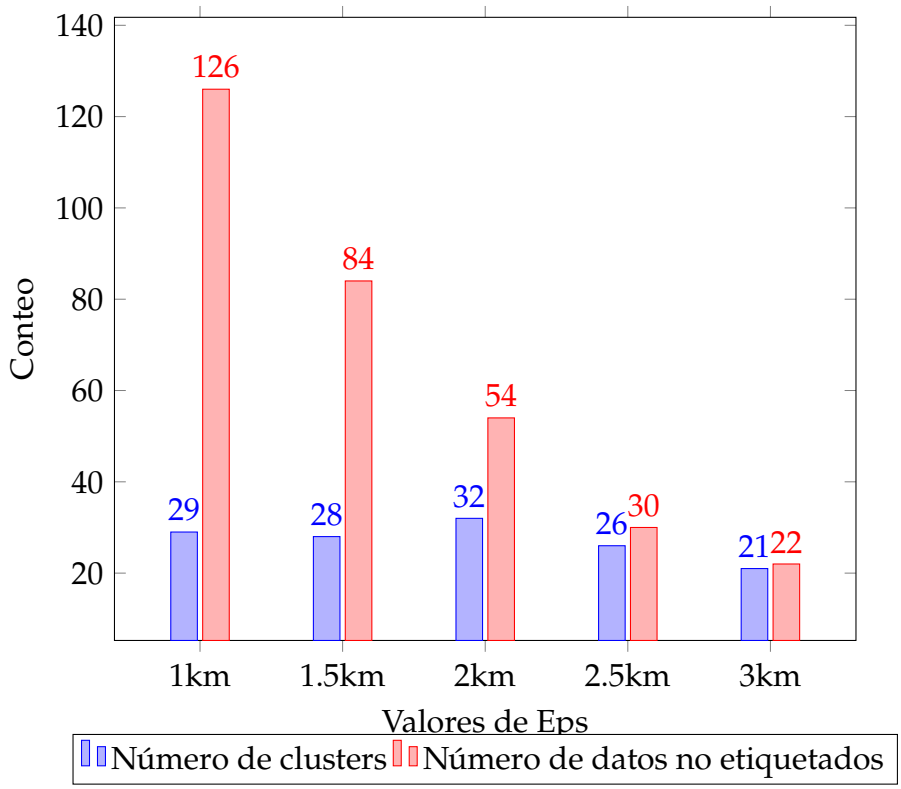
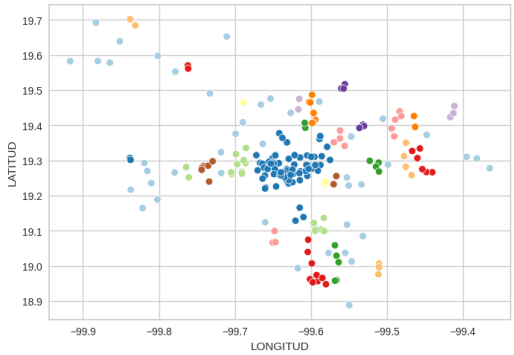
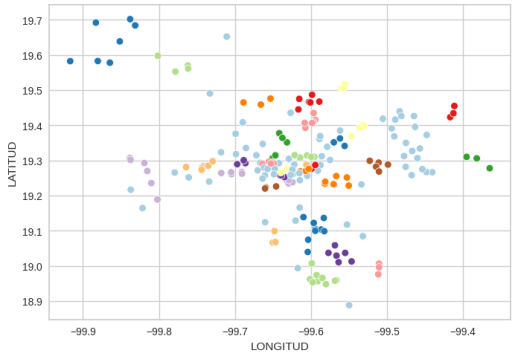


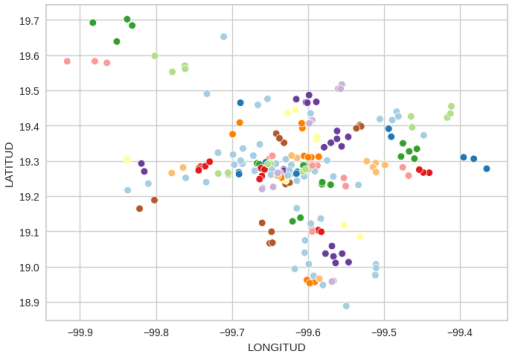
Figura 3: Gráfico de barras de los valores obtenidos al variar Eps



(a) 28 clusters con DBSCAN



(b) 33 clusters con HDBSCAN



(c) 64 clusters con OPTICS

Figura 4: Comparativa de proceso de clustering con los algoritmos basados en densidad

Con el uso de las métricas Silhouette score, índice Davies-Bouldin y el índice Calinski-Harabasz para realizar una comparativa de los cuatro algoritmos de clus-

tering se obtuvieron los siguientes resultados, mostrados en el Cuadro 4, dónde se puede determinar que el mejor algoritmo para establecer clusters, bien formados e identificables, es k-means; pero en un contexto de determinación de paradas vehiculares, no es adecuado por el tamaño y lejanía de los elementos en la frontera del cluster hasta su centroide. Comparando los algoritmos de clustering basados en densidad, el que tienen una mejor valoración para determinar los clusters, es HDBSCAN teniendo mejores valores en cuanto a la calidad de los clusters formados.

Tabla 4: Resultados de las métricas para valorar la calidad de los clusters

Algoritmo	**Silhouette score**	**Índice Davies-Bouldin**	**Índice Calinski-Harabasz**
K-means	0.3894	0.7579	255.6820
DBSCAN	-0.2044	3.6188	2.0485
HDBSCAN	0.2091	1.7195	36.2838
OPTICS	0.3063	2.5802	14.1846

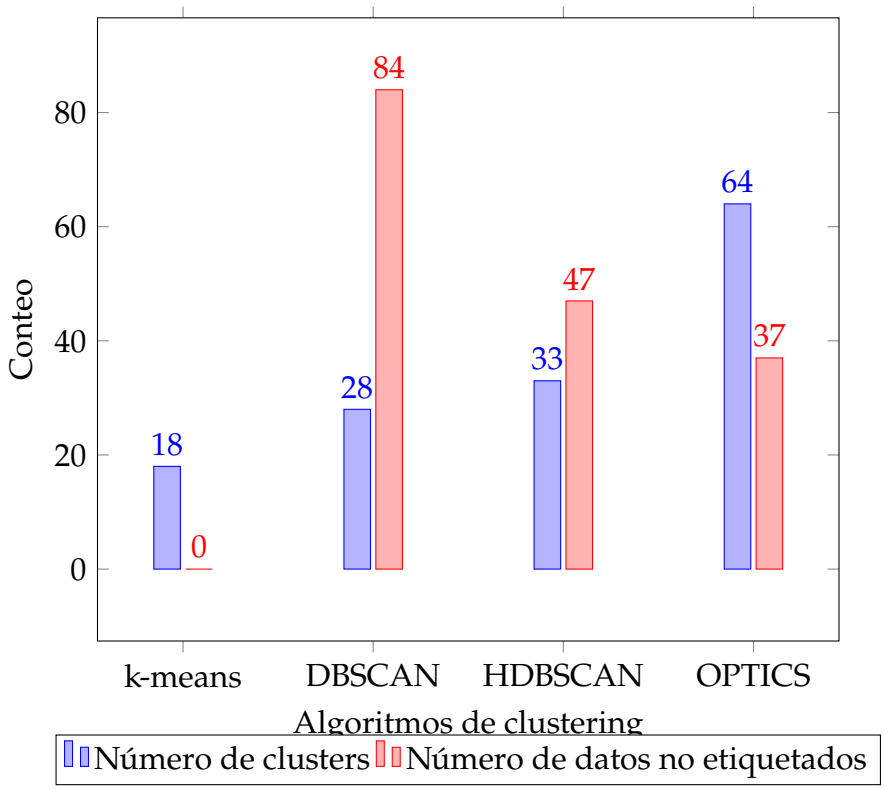


Figura 5: Conteo de clusters y datos no etiquetados por los algoritmos de clustering

En la gráfica de la Figura 5, se muestra la cantidad de clusters y de datos no agrupadas determinandos por cada algoritmo de clustering, de lo cual se puede obtener el total de paradas de autobús de acuerdo a la cantidad de clusters que cada algoritmo identificó.

7. Conclusiones

Los algoritmos de clustering basados en densidad para identificar zonas adecuadas y establecer paradas de autobús utilizando códigos postales son factibles en comparación a algoritmos más recurridos de clustering como k-means. Los algoritmos DBSCAN, HDBSCAN y OPTICS son similares en cuanto a su procesamiento, pero los últimos dos solo requieren un parámetro para su funcionamiento y tienen la característica de calcular el valor adecuado *eps*; esto se demuestra al comparar la cantidad de clusters que pueden identificar, siendo HDBSCAN y OPTICS capaces de determinan una mayor cantidad de clusters de los que puede determinar DBSCAN, sin embargo OPTICS crea una cantidad de clusters mucho mayor pero con problemáticas, en el sentido del contexto de transporte, debido a que los clusters determinados, al ser considerados paradas de autobús, son demasiados y la calidad de los clusters no fue bien valorada por las métricas usadas. El mejor algoritmo basado en densidad es HDBSCAN que permite determinar una cantidad adecuada de clusters y los clusters formados obtienen una buena valoración de calidad en comparación a los otros, consiguiendo así determinar 33 puntos de parada de autobús del conjunto de datos trabajado.

Agradecimientos

Se reconoce el apoyo de la dirección de transporte y a la dirección de control escolar de la Universidad Autónoma del Estado de México, así como los encargados de cada una de las áreas mencionadas por proporcionar los datos necesarios para la realización de las pruebas con el algoritmo de clustering, así como las pruebas necesarias para este trabajo.

Bibliografía

[1] Datascientest. Machine Learning y Clustering: el algoritmo DBSCAN. *Online*, 2023.

[2] A. Géron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. 2019.

[3] Departamento de Matemáticas e Informática de la Universidad de Barcelona. El poder de los datos: Del big data al aprendizaje profundo. *National Geographic*, 2017.

[4] R. Liu, N. Wang. Data-Driven Bus Route Optimization Algorithm Under Sudden Interruption of Public Transport. *IEEE Access*, 10:5250-5263, 2022.

[5] M. Sciortino, R. Lewis, J. Thompson. A School Bus Routing Heuristic Algorithm Allowing Heterogeneous Fleets and Bus Stop Selection. *SN Computer Science*, 4(1):74, 2022.

[6] P. Grabusts. The Choice of Metrics for Clustering Algorithms. *Environment. Technology. Resources. Proceedings of the International Scientific and Practical Conference*, 2:70, 2015.

[7] A. Nagpal, A. Jatain, D. Gaur. Review based on data clustering algorithms. *2013 IEEE CONFERENCE ON INFORMATION AND COMMUNICATION TECHNOLOGIES*, 298–303, 2013.

[8] D. Arthur, S. Vassilvitskii. How slow is the k -means method?. *Proceedings of the twenty-second annual symposium on Computational geometry*, 144–153, 2006.

[9] M. Ester, H. P. Hriegel, J. Sander, X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatian Databases with Npise. *in Proc. AAAI*, 23(1):226–231, 1996.

[10] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, X. Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems*, 42(3):1–21, 2017.

[11] R. J. G. B. Campello, D. Moulavi, A. Zimek, J. Sander. Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. *ACM Transactions on Knowledge Discovery from Data*, 10(1):1–51, 2015.

[12] scikit-learn developers (BSD License). HDBSCAN — scikit-learn.org. *Online*, 2023.

[13] M. Ankerst, M. M. Breunig, H. P. Kriegel. OPTICS: Ordering Points To Identify the Clustering Structure. *ACM SIGMOD Record*, 28(2):49–60, 1999.

[14] scikit-learn developers. OPTICS — scikit-learn.org. *Online*, 2023.

[15] P. Dauni, M. D. Firdaus, R. Asfariani, M. I. N. Saputra, A. A. Hidayat, W. B. Zulfikar. Implementation of Haversine formula for school location tracking. *Journal of Physics: Conference Series*, 1402(7):077028, 2019.

[16] W. Zhang. Applications of the Haversine formula in the evaluation of the great-circle distance between two points on the earth's surface. *Journal of Applied Mathematics and Physics*, 3(7):812–819, 2009.

[17] T. S. Rappaport. Wireless Communications: Principles and Practice. *IEEE Press*, 1996.

[18] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[19] D. L. Davies, D. W. Bouldin. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[20] T. Calinski, J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics - Theory and Methods*, 3(1):1–27, 1974.

[21] Openstreetmap.org. OpenStreetMap. *Online*, 2023.

[22] scikit-learn developers. scikit-learn developers. *Online*, 2023.

Direcciones de correo de los autores

Sánchez Carmona, M. A.: msanchezc048@alumno.uaemex.mx

Informaticae Abstracta
Año 2024, Vol. 2, No. 1, pp. 25-35 ENE | JUN 2024

Importancia de medir la percepción del riesgo ante la contaminación en el Curso Alto del Río Lerma

Hernández Pérez, R; Salinas Tapia, H.; Jiménez Moleón, M. C.

Información del Artículo

Recibido: 3-may-2023
Aceptado: 18-ago-2023

Palabras clave: Percepción del riesgo, Contaminación del agua, Río Lerma, Aguas residuales, Salud pública, Metodología de investigación

Resumen

La Organización Mundial de la Salud [1] estima que más de 505 mil muertes por diarrea al año son causadas por el contacto con agua contaminada. Además, un informe de la UNESCO (2015) indica que el 90 % de las aguas residuales en países en desarrollo se vierten directamente en ríos, lagos o mares sin ningún tipo de tratamiento. En México, la descarga de aguas residuales, tanto municipales como industriales, sin tratamiento previo, es un problema persistente, siendo el sistema hidrográfico Lerma-Chapala-Santiago, uno de los más afectados, con una extensión territorial de 191,500 km^2 y una fuerte contaminación documentada. La presente investigación tiene como objetivo evaluar la percepción del riesgo sobre la contaminación del Curso Alto del Río Lerma (CARL) en el Estado de México. Para ello, se desarrollaron instrumentos de recolección de datos confiables y validados, que se aplicaron a la población ribereña de Tlachaloya, Estado de México, para analizar su conocimiento sobre la contaminación y sus efectos. La metodología incluyó la generación y validación de ítems, la formulación de dimensiones, categorías e indicadores, y la evaluación cuantitativa de los resultados. Los hallazgos revelan una baja percepción del riesgo asociado a las fuentes de contaminación y sus efectos en el río, así como una baja atribución de responsabilidad por parte de la población local, evidenciando un sesgo en el conocimiento científico sobre el tema y la falta de difusión adecuada. Estos resultados subrayan la importancia de utilizar metodologías rigurosas y validadas para identificar la vulnerabilidad de las personas ante la contaminación, y la necesidad de establecer acciones efectivas para mitigar sus efectos.

1. Introducción

El problema ambiental es uno de los desafíos más importantes del mundo actual. Según Del Puerto y Martínez [2], los peligros antrópicos son consecuencia de la acción directa del ser humano sobre la naturaleza, como en caso de la contaminación del agua. La gestión ambiental busca cambios en la sociedad en beneficio de la sustentabilidad [3] Por ello, los problemas tanto de gestión de los recursos naturales y de la sostenibilidad, como de resiliencia social y ecológica de manera interrelacionada, destacan la importancia de aplicar las ciencias sociales a la toma de decisiones sobre la gestión de los recursos naturales [4]. Uno de los enfoque de la percepción del riesgo es examinar la toma de decisiones que se refieren a los hábitos de riesgo realizados por el propio individuo [5].

En México, la Curso Alto del Río Lerma (CARL) tiene un problema histórico de contaminación [6] y, aunque hay estudios sobre riesgos en la zona, aquellos orientados hacia lo que percibe la gente sobre la grave polución que le rodea, son inexistentes.

Es muy importante conocer la percepción de la población en cuanto a riesgos ambientales, además de sus preocupaciones, para planear Programas de Comunicación de Riesgos que sean favorables para la participación comunitaria y que permitan lograr cambios de comportamiento en la población. Estos cambios de comportamiento deberán tener como consecuencia la disminución de la exposición a los riesgos ambientales [7].

El objetivo de esta investigación es demostrar la importancia de la percepción del riesgo sobre la contaminación del CARL, en el Estado de México, mediante el planteamiento de instrumentos de recolección de datos confiables y validados, para contrastar la información con el grado de conocimiento que la población tiene respecto a la contaminación y sus diferentes efectos. Lo anterior, a través de la aplicación de un instrumento de medición en la población ribereña de Tlachaloya, Estado de México.

2. Métodología

Antes de desarrollar el cuestionario, se realizó una visita a la comunidad de Tlachaloya, comunidad de Toluca, México, para establecer una relación con las autoridades de la Escuela Primaria Profesor Francisco Estrada, ubicada en el poblado de Tlachaloya Primera Sección, localidad asentada a pocos metros del río Lerma, en el Curso Alto, con la finalidad de recibir el consentimiento de los padres de familia para llevar a cabo el estudio (Figura 1). Los participantes del estudio se mantuvieron en el anonimato y firmaron un formato de consentimiento informado.

Figura 1: Zona de estudio: Curso Alto del Río Lerma

Para la construcción del cuestionario se llevó a cabo una recopilación de instrumentos publicados en diversas bases de datos y plataformas académicas digitales, para revisar los criterios ya evaluados de otros autores y así reunir grupos de conceptos similares al tema relacionado con la percepción ante la contaminación de ríos.

Una vez que se diseñaron las dimensiones y atributos, además de los ítems, se seleccionó un panel de expertos, al quien se envió el formato de validación que abarcó diversos elementos de identificación como: objetivo, población, importancia del instrumento, forma de responder a la prueba e instrucciones para evaluar. También incluyeron las dimensiones, categorías, indicadores y reactivos, además de los aspectos a evaluar. La retroalimentación de los jueces se recabó junto con una constancia de validación y formato de identificación profesional. Finalmente,

y, con base en esas observaciones, se realizaron los ajustes del instrumento (Figura 2).

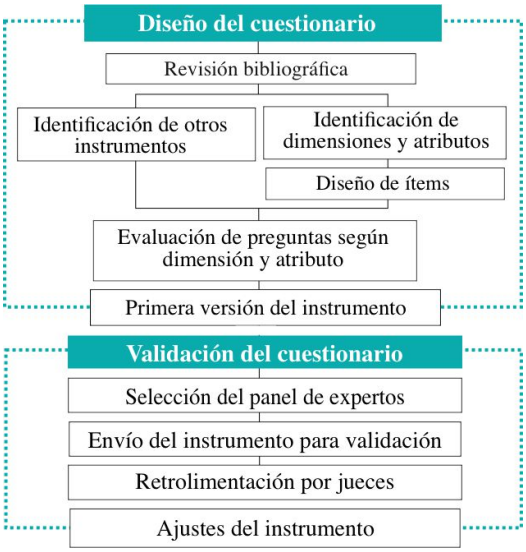


Figura 2: Etapas del diseño del instrumento y la validación por expertos [8]

La aplicación del cuestionario se realizó de manera presencial, con la ayuda del *Software MimioStudio 11.55* (*Machintosh)* para el diseño de preguntas, con dos sistemas interactivos de respuesta *MimioVote* y con 64 dispositivos portátiles para responder preguntas de manera remota.

Se realizó un análisis cuantitativo buscando correlaciones (anti-imagen), fiabilidad (α de Cronbach)y validez del instrumento (Esfericidad de Bartlet y el índice KMO), cuyos resultados se procesaron tanto en SPSS, como en Excel (Figura 3).

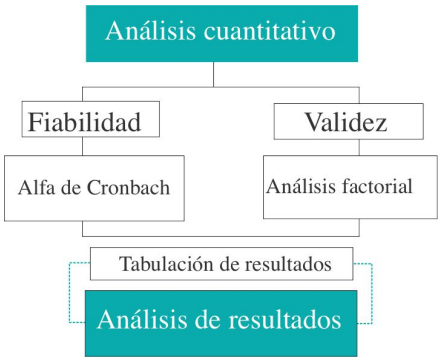


Figura 3: Validación cuantitativa

El coeficiente de consistencia interna dependió directamente de las correlaciones entre los ítems o reactivos, esto es, del grado en que los ítems midieron la misma variable. Mientras más homogeneidad entre los ítems, mayor fue el valor de la consistencia interna [9]. El α es la medida en que los examinados se inclinaron a contestar correctamente cada ítem y se utilizó porque suele aplicarse a reactivos de respuestas politómicas para examinar la fiabilidad, a través de las correlaciones entre los ítems y el total; su consistencia interna posee un rango entre cero y uno, y su valor debe ser > 0.7 para respaldar su confiabilidad [10]. En este sentido, el α se calculó de forma general en SPSS para todo el instrumento y además se obtuvo

un valor por cada dimensión, con la finalidad de suprimir los ítems cuyos valores estaban por debajo del 0.7.

Con los grupos de ítems y dimensiones ya organizados, se realizó un análisis confirmatorio para corroborar la agrupación de preguntas que se planteó originalmente. El objetivo de este análisis fue reducir dimensiones e ítems para garantizar la validez del instrumento [11] y hacerlo más fácilmente interpretable [12]. Con este análisis se buscaron grupos homogéneos de variables, a partir de un conjunto numeroso de las mismas pues, según [13], se debe procurar que cada grupo (o dimensión) sea independientes del otro. Por ello, se buscó el índice de correlaciones por dimensión, realizando un análisis de cada una de manera independiente. Esto permitió definir cuántos factores se esperaban, qué factores estaban relacionados entre sí, y qué ítems estaban relacionados con cada factor [12].

La alta correlación entre variables debe ser comprobada, para confirmar la pertinencia del análisis, ya que la falta de asociación entre ellas imposibilita la realización del análisis factorial [12]. Dicha confirmación se llevó a cabo a través del Análisis de la Matriz de Correlación cuyos indicadores fueron:

a) Matriz de Correlación Anti-imagen, donde se tomaron los valores superiores a 0.5 en la diagonal principal de la matriz, como el mínimo indicativo para proceder al análisis [14].

b) Índice de Kaiser Meyer Olkim (KMO), el cual se usó para comparar las magnitudes de los coeficientes de correlación parcial [12]. Por ello valores entre 0.5 y 1 indicaron la pertinencia de aplicarlo [14]. Si el resultado fue $\geqslant 0.9$, el test fue muy bueno; $\geqslant 0.8$, notable; $\geqslant 0.7$, mediano; $\geqslant 0.6$, bajo y, < 0.5, muy bajo (UA, 2016-2020) y, en consecuencia, sería menos apropiado realizar al Análisis Factorial.

c) Prueba de esfericidad de Bartlett cuyos resultados permitieron evaluar la hipótesis nula que afirmaba que las variables no estaban correlacionadas. Al rechazar la hipótesis, se consideró que las variables estaban los suficientemente inter correlacionadas para realizar el análisis [15], por lo que se considera adecuado con un nivel de significación menor de 0.05.

d) Criterio de determinación *a priori* para realizar la extracción de factores hasta llegar a un buen ajuste [16, 17].

3. Resultados

La primera versión del instrumento de medición consistió en 50 preguntas de opción múltiple que, con los ajustes del jueceo, se redujeron a 30 de las cuales 25 fueron sobre percepción del riesgo y cinco sobre datos demográficos (sexo, edad, estado civil, ocupación y escolaridad). La percepción del riesgo evaluó la información y las experiencias de riesgo acumuladas por las personas, con preguntas como: percepción de la pérdida de biodiversidad, percepción con relación a la salud y percepción de riesgos personales.

Los jueces fueron 11 especialistas tanto de química del agua, como de gestión integrada de los recursos hídricos, hidráulica, monitoreo y calidad ambiental, investigación en recursos hídricos, asesoría gubernamental del río Lerma en agua y saneamiento, filosofía ambiental y análisis de medios de comunicación.

Tabla 1: Ajustes de la validación por jueces.

Dimensiones	No. de preguntas	Categorías
Nivel de percepción general del riesgo	4 20 6	Percepción del riesgo ambiental Percepción en relación con la salud Percepción de riesgos personales
Datos demográficos	5	Sexo, edad, estado civil, ocupación, escolaridad.
Total	35	Total

Según los resultados, el total de encuestados originalmente fue de 64 padres de familia o tutores de alumnos de nivel primaria, sin embargo, el número se redujo a 55, debido a pequeñas fallas en tres equipos, además de que seis usuarios no respondieron sistemáticamente la encuesta.

De las 55 personas participantes, 43.6 % tenían entre 31 a 35 años, y 38.2 % entre 18 y 30; se trataba en su mayoría de personas del sexo femenino (81.8 %), casadas (69.1 %), con un nivel educativo de primaria (61.8 %) y sus labores estaban dedicadas principalmente al hogar (65.5 %) o al comercio (14.5 %). Del mismo modo, en cuanto al apego territorial, el 51 % de los encuestados respondió ser originario de Tlachaloya, con padres oriundos del mismo lugar (59 %). El 91 % señaló como lugar de residencia el primer sector, esto es, la zona más cercana al río Lerma y el 54.5 % expresó habitar en casa propia.

Para calcular la confiabilidad y consistencia interna se determinó el coeficiente α de Cronbach, el cual fue de 0.958, resultado considerado excelente para este tipo de estudios [18]. Del mismo modo, al obtener el índice KMO, este arrojó un resultado de 0.862, considerado notable para realizar este análisis, mientras que la prueba de Esfericidad de Bartlet trajo como resultado el valor más confiable de 0.000. Con esto se confirmó que esta dimensión, además de fiabilidad, también tuvo validez. En cuanto a los resultados de la matriz de Correlación Anti-Imagen, en la diagonal se observaron valores > 0.774, denotando una buena adecuación muestral entre las variables.

La Figura 4 muestra un grado bajo de percepción de fuentes de contaminación, al considerar que ni la ganadería ni las tormentas afectan negativamente a la calidad del agua; incluso el 32 % y el 26 % desconoce dichos efectos.

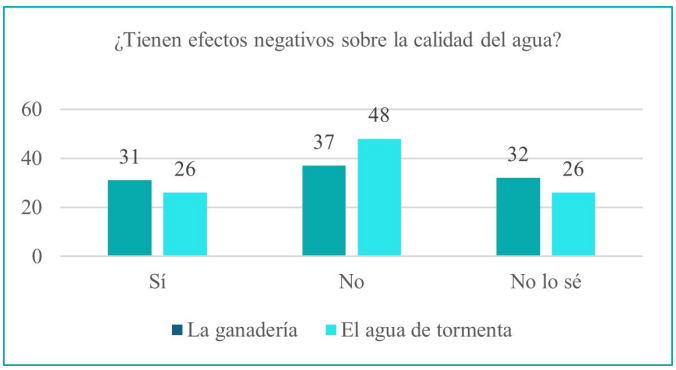


Figura 4: Percepción de fuentes de contaminación.

La población encuestada está consciente de la problemática que vive, por lo que muestra una preocupación importante por la contaminación del agua en su comunidad (Figura 5).

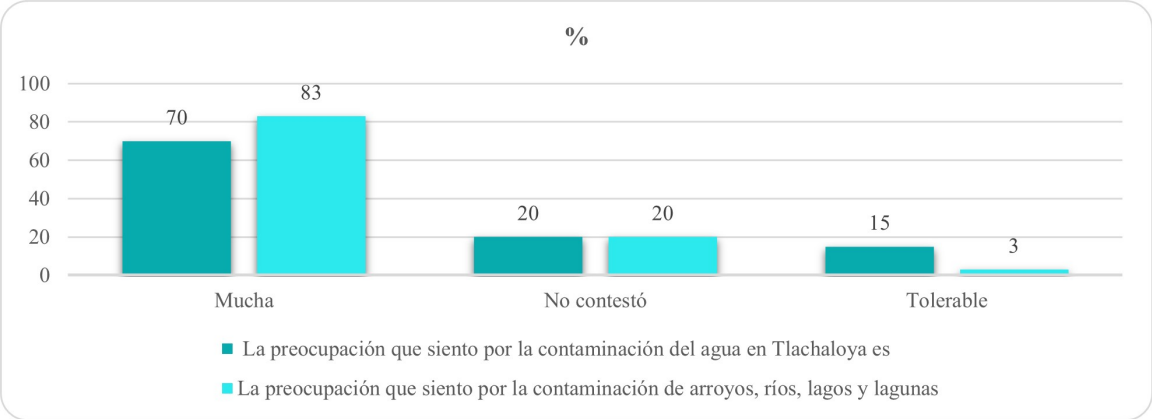


Figura 5: Percepción de riesgos por la contaminación del agua.

En la Figura 6, los encuestados refieren un desconocimiento en relación con la comunidad más afectada por la contaminación del río Lerma, aunque el 22 % de ellos consideran que Tlachaloya tiene importantes niveles de contaminación, incluso mucho mayor que las otras comunidades.

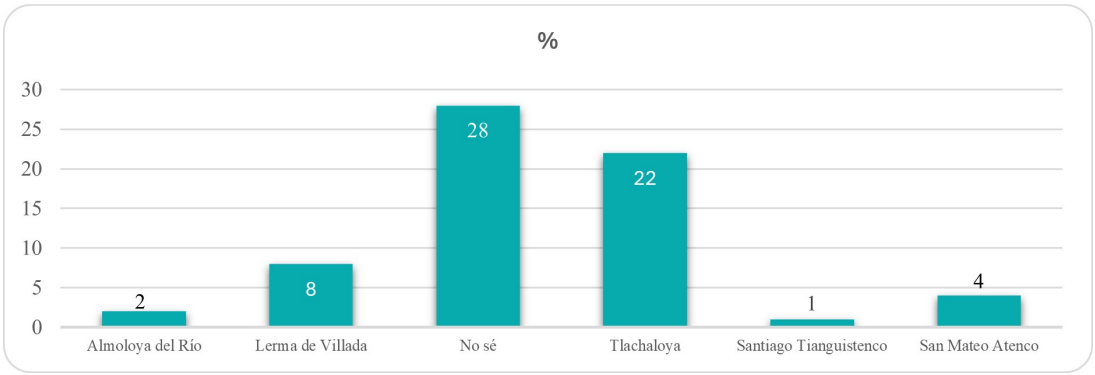


Figura 6: La localidad más afectada por la contaminación del río Lerma

Al cuestionarles sobre el uso que podrían darle al agua del río, partiendo desde el uso doméstico hasta recreativo, reflejaron una falta de integración con éste, pues el 72 % respondió que nunca realizaría ninguna actividad en éste. Esto refleja el conocimiento vivencial que tienen sobre las condiciones del río (Figura 7).

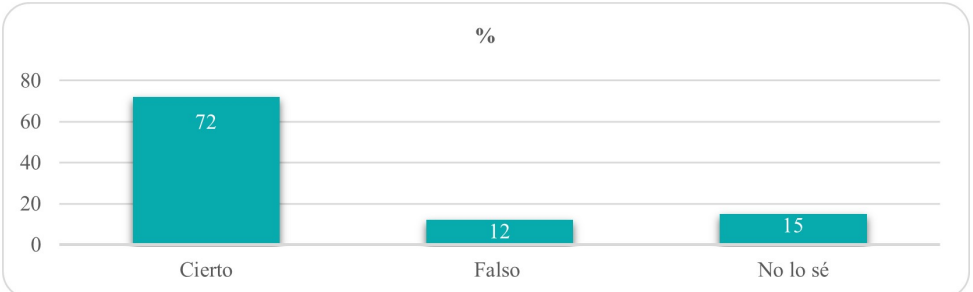


Figura 7: Nunca usaría el agua del río Lerma, en ninguna actividad.

La Figura 8 muestra un nivel elevado de afectación por las condiciones del río Lerma, por lo que su grado de percepción del riesgo en esta parte pareciera indudable.

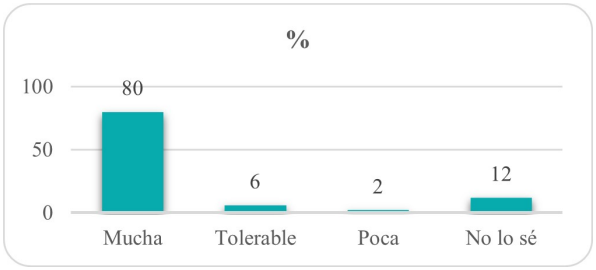


Figura 8: Afectación del encuestado por la contaminación del río.

En cuanto a los problemas de salud, se plantearon algunas enfermedades derivadas del agua contaminada y se observó que los encuestados afirmaron estar enterados de los padecimientos a los que están expuestos por el río. No obstante, la mayoría de ellos también refirieron desconocimiento ante ciertas enfermedades, especialmente en lo que se refiere al síndrome del bebé azul y a los problemas reproductivos (Figura 9).

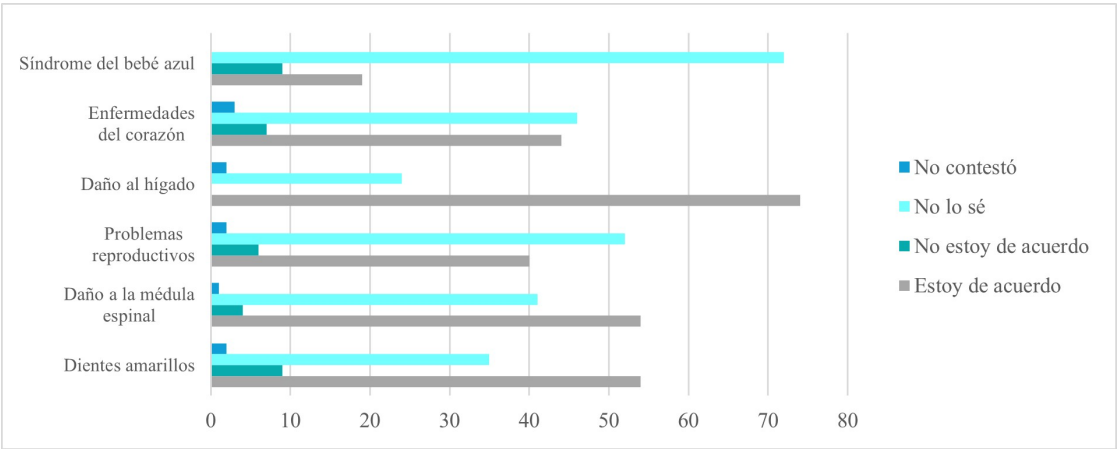


Figura 9: Percepción del riesgo en la salud (posibles enfermedades causadas por la contaminación del río Lerma)

Según Colciencias (2010), la apropiación social del conocimiento es un proceso de comprensión e intervención de las relaciones entre tecnociencia y sociedad, construido con la participación de los diversos grupos sociales. En este sentido, no basta con que el individuo afirme tener una percepción del riesgo elevada, si no genera cambios de comportamiento que lo lleven a modificar sus hábitos. Esta apropiación puede manejarse a través de las potencialidades de la divulgación científica [19].

Desde esta perspectiva, la Figura 10 evidencia la falta de acciones para solucionar la problemática de contaminación, cuya preocupación se mostró elevada en respuestas anteriores. Por ello, es posible observar una muy baja participación en campañas ambientales del municipio o incluso poca cooperación económica para recuperar el río, presentando una incongruencia en la percepción del riesgo expresado preliminarmente.

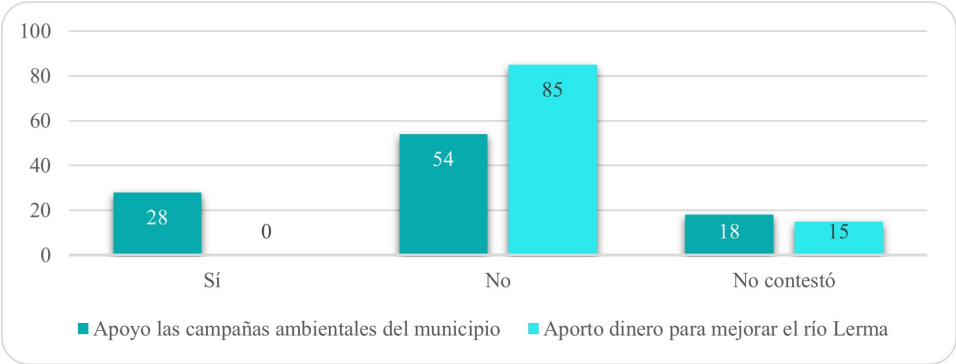


Figura 10: Acciones de eficiencia personal que apoyen la percepción del riesgo.

Del mismo modo, la Figura 11 muestra que los encuestados consideran responsables de la contaminación de ríos a otros actores como la industria o la falta de legislación, pero exime a los hogares de esta obligación.

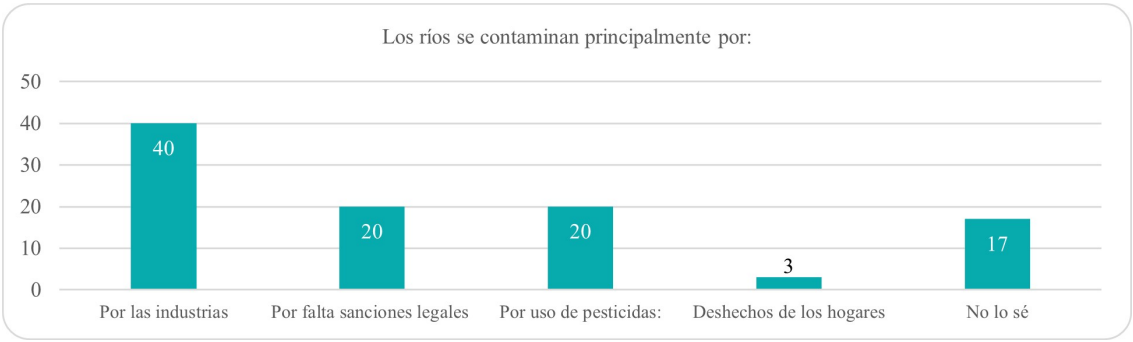


Figura 11: Atribución de responsabilidades

Según [20], los cambios que se presentan a nivel sociocultural, los desarrollos científicos y el nivel sociopolítico y socioeconómico, únicamente se pueden entender de modo interdisciplinario. Clark y Wallace (2015) afirman que, en el mundo actual, que permite un fácil acceso a todas las formas de conocimiento a través de los medios electrónicos, lo anterior es muy posible de lograr. En este sentido, el conocimiento resulta por demás importante para que exista una percepción correcta del riesgo. La Figura 12 refleja lo que la población encuestada piensa en torno a la calidad del agua de un manantial, al cual consideran permanentemente limpio, por lo que no cuentan con información certera sobre este tema en particular.

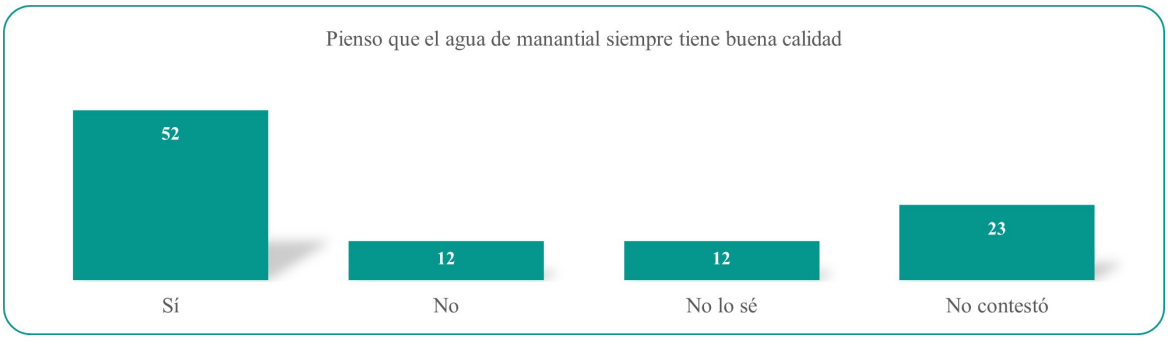


Figura 12: Conocimiento sobre la calidad del agua de un manantial

Lo anterior demuestra la importancia de realizar estudios de percepción para constatar el grado de conocimiento científico que la población tiene y en su caso,

diseñar campañas de divulgación científica que promuevan una apropiación del conocimiento que originen un cambio de actitud ante los problemas ambientales.

4. Conclusiones

- La validación por expertos y psicométrica, permitieron definir las categorías e ítems correctos del instrumento de medición.
- El uso de *Mimio Studio* (*Mimi Vote)* agilizó de manera importante la aplicación y tratamiento de datos del instrumento de medición.
- El índice general de Alfa de Cronbach mostró valores excelentes. La medida KMO y la prueba de esfericidad de Bartlett revelaron buenas correlaciones entre las variables.
- Los resultados reflejaron una baja percepción de las fuentes de contaminación del agua, un bajo compromiso, un desconocimiento en cuanto a algunas enfermedades a consecuencia del agua contaminada y poca responsabilidad para asumir la protección y preservación del medio ambiente.
- Es importante realizar estudios de percepción del riesgo para diseñar programas de divulgación científica y promover políticas públicas en torno a la contaminación de ríos.

Bibliografía

[1] Organización Mundial de la Salud. Agua para consumo humano. 2023. Recuperado de `https://acortar.link/0jSHp`

[2] J.A. Del Puerto y Y. Martínez. Peligros ambientales y antrópicos sobre las aguas de la Comuna de Ondjiva, Angola. *riha* [online], vol. 42, n. 3, pp. 22-46, 2021. ISSN 1680-0338.

[3] J.J. Sánchez, J.M. Segovia y P. Sánchez. *Metodología de la Investigación Social, técnicas innovadoras y sus aplicaciones*. Madrid: Editorial Síntesis, 2012.

[4] S. Charnleya, C. Carothersb, T. Satterfieldc, A. Levined, M. Poee, K. Normane, J. Donatutof, S.J. Breslowe, M.B. Masciag, P.S. Levine, X. Basurtoh, Ch.C. Hicksi, J.C. García-Quijanok, K. St. Martin. Evaluating the best available social science for natural resource management decision-making. *Environmental Science and Policy*, 73, pp. 80-88, 2017.

[5] M. Stanojlovic. Percepción social de riesgo: una mirada general y aplicación a la comunicación de salud. *Revista de Comunicación y Salud*, Vol. 5, ISSN: 2174-5323, pp. 96-107, 2015.

[6] H. Salinas, L. Flores, J.A. García, S. Tejeda y B. López. Modelación del Curso Alto del Río Lerma (CARL), utilizando QUAL2Kw, considerando la distribución y variación de nitrógeno amoniacal y nitrógeno como nitratos. *Aqua-LAC*, Vol. 8, Nº 1, pp. 34-43, 2016.

[7] R. Torres-Nerio, G. Domínguez-Cortinas, A. van't Hooft, F. Díaz-Barriga Martínez, y A.C. Cubillas-Tejeda. Análisis de la percepción de la exposición a riesgos ambientales para la salud, en dos poblaciones infantiles, mediante la elaboración de dibujos. *Salud Colectiva*, vol. 6, n. 1, pp. 65-81, 2010.

[8] M.J. Letelier, M.B. Aller, D. Henao, et al. Diseño y validación de un cuestionario para medir la continuidad asistencial entre niveles desde la perspectiva del usuario: CCAENA. *Gac Sanit*, 24(4), pp. 339-346, 2010.

[9] M. Quero. Confiabilidad y coeficiente Alpha de Cronbach. *Telos*, vol. 12, núm. 2, pp. 248-252, 2010.

[10] H. Sánchez, C. Reyes y K. Mejía. *Manual de términos en investigación científica, tecnológica y humanística*. Universidad Ricardo Palma, 2018. ISBN 978-612-47351-4-1.

[11] J. Supo. *Cómo validar un instrumento – La guía para validar un instrumento en 10 pasos*. Biblioteca Nacional del Perú, 2013.

[12] S. De la Fuente. *Análisis factorial.* Universidad Autónoma de Madrid, 2011. Recuperado de `https://bit.ly/3c0WcaA`

[13] S.A. Ochoa y J.A. Toscano. Revisión crítica de la literatura sobre el análisis financiero de las empresas. *Nóesis. Revista de Ciencias Sociales y Humanidades*, Vol. 21, No 41, pp. 73-99, 2012.

[14] O. Montoya. Aplicación del análisis factorial a la investigación de mercados. Caso de estudio. *Scientia et Technica*, Año XIII, No 35, ISSN 0122-1701, pp. 281-286, 2007.

[15] E. R. Pérez y L. Medrano. Análisis Factorial Exploratorio: Bases Conceptuales y Metodológicas. *Revista Argentina de Ciencias del Comportamiento*, Vol. 2, N°1, ISSN 1852-4206, pp. 58-66, 2010.

[16] S. Zamora, L. Monroy y C. Chávez. *Análisis factorial: una técnica para evaluar la dimensionalidad de las pruebas*. Cuaderno técnico 6. México: Ceneval, 2009.

[17] S. Lloret-Segura, A. Ferreres-Traver, A. Hernández-Baeza y I. Tomás-Marco. El análisis factorial exploratorio de los ítems: una guía práctica, revisada y actualizada. *Anales de psicología*, vol. 30, no 3, pp. 1151-1169, 2014. Recuperado de `http://scielo.isciii.es/pdf/ap/v30n3/metodologia1.pdf`

[18] A. Carvajal, C. Centeno, R. Watson, M. Martínez, y Á. Sanz. ¿Cómo validar un instrumento de medida de la salud? *An. Sist. Sanit. Navar.*, Vol. 34, No 1, pp. 63-72, 2011.

[19] L. Dávila. Divulgación para la apropiación del conocimiento científico y tecnológico. Caracterización y propuesta de estudio. 2020.

[20] K. Das y B. Paital. Future call for policy making to speed up interdisciplinarity between natural and social sciences and humanities in countries such as India. *Heliyon*, 7, pp. 1-18, 2021. Recuperado de `https://acortar.link/WGR9NL`

Direcciones de correo de los autores

Hernández Pérez, R: rhp@uaemex.mx

Salinas Tapia, H.: hsalinast@uaemex.mx

Jiménez Moleón, M. C.: mcjimenezm@uaemex.mx

Informaticae Abstracta
Año 2024, Vol. 2, No. 1, pp. 36-72 JUL | DIC 2024

Concurrencia en Python: Teoría y una Aplicación para la Gestión del Portapapeles de Microsoft Windows

Salas-García, Javier; Portillo-Rodríguez, Otniel; Valle-Cruz, David; Moran-Gonzalez, Miguel Armando

Información del Artículo

Recibido: 10-sep-2023
Aceptado: 18-dic-2023

Palabras clave: threading, python, sincronización de hilos, portapapeles mejorado

Resumen

Este artículo presenta un análisis del «threading» en Python, abordando su implementación, optimización y aplicaciones prácticas en el desarrollo de software moderno. Partiendo de los fundamentos de la programación concurrente, se examina el modelo de concurrencia específico de Python, incluyendo las implicaciones del Global Interpreter Lock (GIL). El estudio profundiza en las herramientas de sincronización como locks, semáforos, eventos y colas, proporcionando ejemplos prácticos de su aplicación. Se analizan las situaciones en las que el «threading» mejora significativamente el rendimiento, particularmente en operaciones intensivas de E/S y procesamiento paralelo de datos. El artículo culmina con un caso de estudio detallado de una aplicación de visor de historial del portapapeles de Microsoft Windows, demostrando la aplicación práctica de los conceptos discutidos. Este artículo ofrece puntos tanto para desarrolladores novatos como experimentados, contribuyendo al entendimiento y la implementación efectiva de la programación concurrente en Python.

1. Introducción

En el campo de la ciencia de la computación, coexisten dos paradigmas fundamentales de ejecución de programas: la programación secuencial y la programación paralela. Estos enfoques difieren significativamente en su metodología de ejecución de instrucciones y en su capacidad para utilizar los recursos computacionales disponibles [1].

La programación secuencial, también denominada programación lineal, se caracteriza por la ejecución de instrucciones en un orden estrictamente definido. En este modelo, cada instrucción se procesa de manera individual y consecutiva, siguiendo una secuencia predeterminada [2].

El tiempo de ejecución total de un programa secuencial (T_s) puede representarse como la suma de los tiempos de ejecución de cada instrucción individual:

$$T_s = \sum_{i=1}^{n} t_i \tag{1}$$

donde n es el número total de instrucciones y t_i es el tiempo de ejecución de la i-ésima instrucción.

En contraste, la programación paralela permite la ejecución simultánea de múltiples instrucciones o secuencias de instrucciones. Este enfoque se basa en el principio de descomposición de un problema en subproblemas que pueden resolverse concurrentemente [3].

El tiempo de ejecución de un programa paralelo (T_p) puede modelarse de la siguiente manera:

$$T_p = T_s/p + T_o \tag{2}$$

donde p es el número de procesadores y T_o es el tiempo de sobrecarga debido a la comunicación y sincronización entre procesos.

La eficiencia de la paralelización se puede cuantificar mediante la ley de Amdahl [4], que establece que la aceleración máxima (S) que se puede lograr está limitada por la fracción del programa que no se puede paralelizar:

$$S = \frac{1}{(1-f) + f/p} \tag{3}$$

donde f es la fracción del programa que se puede paralelizar y p es el número de procesadores.

En el contexto específico de Python, el módulo «threading» proporciona una interfaz para implementar programación paralela a nivel de hilos. Aunque el Intérprete Global de Python (GIL) impone ciertas limitaciones en la ejecución verdaderamente paralela de hilos en operaciones intensivas de CPU, el «threading» en Python sigue siendo una herramienta valiosa para mejorar el rendimiento en tareas limitadas por operaciones de entrada/salida [5].

De este modo, la elección entre programación secuencial y paralela depende de factores como la naturaleza del problema, la arquitectura del sistema y los requisitos de rendimiento. Un análisis cuidadoso de estos factores es esencial para determinar el enfoque más adecuado en cada situación específica.

2. Importancia del «Threading» en la Era de Procesadores Multinúcleo

La proliferación de procesadores multinúcleo ha revolucionado el panorama de la computación, haciendo que el «threading» se convierta en un aspecto crucial del desarrollo de software moderno. El «threading», como implementación de la programación concurrente a nivel de software, permite aprovechar eficazmente la capacidad de procesamiento paralelo que ofrecen estas arquitecturas avanzadas [6].

En el contexto de los sistemas multinúcleo, el «threading» adquiere una relevancia particular por varias razones:

1. **Utilización eficiente de recursos:** El «threading» permite que múltiples tareas se ejecuten simultáneamente en diferentes núcleos, maximizando la utilización de los recursos de hardware disponibles. Esto se traduce en una mejora significativa del rendimiento, especialmente en aplicaciones que manejan cargas de trabajo intensivas [7].

2. **Mejora de la capacidad de respuesta:** En aplicaciones interactivas, el «threading» puede mejorar significativamente la experiencia del usuario al permitir que las operaciones de larga duración se ejecuten en segundo plano, manteniendo la interfaz de usuario receptiva [8].

3. **Escalabilidad:** A medida que aumenta el número de núcleos en los procesadores modernos, las aplicaciones que utilizan «threading» de manera efectiva pueden escalar su rendimiento de forma casi lineal, un fenómeno conocido como escalabilidad de Gustafson [9].

4. **Eficiencia energética:** El «threading» permite una distribución más uniforme de la carga de trabajo entre los núcleos, lo que puede conducir a una mejor gestión térmica y eficiencia energética en comparación con la ejecución en un solo núcleo a alta frecuencia [10].

Sin embargo, es importante destacar que el «threading» también introduce desafíos significativos. La sincronización y comunicación entre hilos pueden ser complejas y propensas a errores, como las condiciones de carrera y los interbloqueos. Además, en lenguajes como Python, el Global Interpreter Lock (GIL) puede limitar los beneficios del «threading» en operaciones intensivas de CPU [5].

A pesar de estos desafíos, el «threading» sigue siendo una herramienta fundamental para el desarrollo de software eficiente en la era de los procesadores multinúcleo. Su importancia se magnifica en aplicaciones que requieren un alto grado de concurrencia, como servidores web, sistemas de bases de datos y aplicaciones de procesamiento de datos a gran escala.

3. Beneficios del «Threading» para la Optimización de Programas

El «threading», como técnica de programación concurrente, ofrece una serie de beneficios significativos para la optimización de programas. Estos beneficios se manifiestan en diversos aspectos del rendimiento y la eficiencia del software, especialmente en el contexto de sistemas con múltiples núcleos de procesamiento. A continuación se describen algunos de estos beneficios.

1. Mejora del tiempo de respuesta: Uno de los beneficios más notables del «threading» es la mejora en el tiempo de respuesta de las aplicaciones. Al permitir que múltiples tareas se ejecuten concurrentemente, el «threading» puede reducir significativamente el tiempo de espera del usuario.

2. Aumento del «throughput»: El «threading» puede aumentar el throughput de un programa, es decir, la cantidad de trabajo realizado por unidad de tiempo. En sistemas con múltiples procesadores o núcleos, los hilos pueden distribuirse entre estos recursos de hardware, permitiendo que se realicen más operaciones simultáneamente. Este paralelismo puede llevar a mejoras sustanciales en el rendimiento, especialmente en aplicaciones que manejan grandes volúmenes de datos o cálculos complejos [11].

3. Modelo de programación más «natural» para ciertos problemas: Para ciertos tipos de problemas, el «threading» proporciona un modelo de programación más natural y fácil de conceptualizar. Por ejemplo, en simulaciones de sistemas físicos o en modelado de entidades independientes, cada entidad puede

representarse como un hilo separado, lo que facilita la implementación y el mantenimiento del código [7].

4. Escalabilidad mejorada: El «threading» permite que las aplicaciones escalen de manera más eficiente con el aumento de los recursos de hardware. A medida que se añaden más núcleos o procesadores a un sistema, las aplicaciones bien diseñadas que utilizan «threading» pueden aprovechar automáticamente estos recursos adicionales sin necesidad de modificaciones significativas en el código [12].

5. Latencia reducida en sistemas distribuidos: En sistemas distribuidos, el «threading» puede ayudar a reducir la latencia al permitir que múltiples operaciones de red se realicen simultáneamente. Esto es especialmente beneficioso en aplicaciones cliente-servidor, donde múltiples solicitudes pueden procesarse en paralelo, mejorando el tiempo de respuesta global del sistema [13].

6. Optimización del Uso de Caché: El «threading» puede llevar a una mejor utilización de la memoria caché del procesador. Al dividir un problema en subproblemas más pequeños que se ejecutan en hilos separados, es posible que cada hilo opere en un conjunto de datos que cabe completamente en la caché, reduciendo los costosos accesos a la memoria principal [14].

Estos beneficios del «threading» contribuyen significativamente a la optimización de programas, permitiendo un mejor aprovechamiento de los recursos de hardware modernos y mejorando el rendimiento general de las aplicaciones en diversos escenarios.

4. Fundamentos del «Threading»

En el ámbito de la programación concurrente, el concepto de thread (hilo en español) es fundamental. Un thread puede definirse como la unidad más pequeña de ejecución dentro de un proceso que puede ser gestionada independientemente por el planificador de un sistema operativo [15]. En términos más sencillos, un thread representa una secuencia única de instrucciones que pueden ejecutarse de forma paralela con otras secuencias dentro del mismo programa.

Para comprender mejor este concepto, se puede imaginar un thread como un trabajador individual en una fábrica. Así como múltiples trabajadores pueden realizar diferentes tareas simultáneamente en una línea de producción, varios threads pueden ejecutar diferentes partes de un programa al mismo tiempo. Cada thread tiene su propio contador de programa, pila de ejecución y conjunto de registros, pero comparte el espacio de memoria y los recursos del proceso principal con otros threads [16, 17].

En el contexto de la programación, los threads se utilizan para lograr la concurrencia, permitiendo que múltiples operaciones se realicen simultáneamente. Esto es especialmente útil en sistemas con múltiples núcleos de procesamiento, donde diferentes threads pueden ejecutarse en paralelo en diferentes núcleos, mejorando potencialmente el rendimiento general del programa [11].

Es importante destacar que la creación y gestión de threads conlleva cierta sobrecarga computacional. Cada thread requiere recursos del sistema, como espacio en la pila y tiempo de CPU para la conmutación de contexto. Por lo tanto, el uso eficiente de threads implica un equilibrio entre el número de threads creados y los beneficios de rendimiento obtenidos [8].

En lenguajes de programación modernos como Python, Java o C++, existen bibliotecas y módulos específicos que facilitan la creación y gestión de threads. Estos proporcionan abstracciones de alto nivel que permiten a los desarrolladores implementar programas concurrentes sin necesidad de manejar directamente los detalles de bajo nivel de la creación y sincronización de threads [5].

En resumen, un thread es una unidad de ejecución ligera dentro de un proceso que permite la concurrencia en programas. Su comprensión y uso adecuado son muy importantes para el desarrollo de software eficiente y de alto rendimiento en la era de los sistemas multinúcleo y las aplicaciones complejas.

5. Diferencias entre Procesos y Threads

Para comprender plenamente el concepto de «threading», es crucial establecer una distinción clara entre procesos y threads. Aunque ambos son unidades de ejecución, difieren significativamente en su estructura, gestión de recursos y propósito dentro de un sistema operativo.

Un proceso, en el contexto de sistemas operativos, representa una instancia de un programa en ejecución. Cada proceso tiene su propio espacio de direcciones de memoria, lo que significa que opera de manera aislada y no puede acceder directamente a la memoria de otros procesos. Esta característica proporciona un alto grado de protección y aislamiento entre diferentes aplicaciones en ejecución [16].

En contraste, los threads existen dentro del contexto de un proceso. Múltiples threads de un mismo proceso comparten el espacio de direcciones de memoria y otros recursos del proceso padre. Esta característica permite una comunicación más eficiente entre threads, pero también requiere mecanismos de sincronización para evitar conflictos en el acceso a datos compartidos [17].

La creación y gestión de procesos implica una sobrecarga significativa para el sistema operativo. Cada nuevo proceso requiere la asignación de un nuevo espacio de direcciones, la inicialización de estructuras de datos del kernel y la copia de recursos del proceso padre. Por otro lado, la creación de threads es más ligera, ya que comparten los recursos del proceso existente, lo que resulta en un menor tiempo de creación y cambio de contexto [15].

En términos de comunicación, los procesos requieren mecanismos de comunicación entre procesos (IPC) proporcionados por el sistema operativo, como tuberías, colas de mensajes o memoria compartida. Estos mecanismos son relativamente costosos en términos de rendimiento. Los threads, al compartir el mismo espacio de memoria, pueden comunicarse de manera más eficiente simplemente accediendo a variables compartidas, aunque esto introduce la necesidad de sincronización [18].

La robustez es otra área de diferencia. Un fallo en un thread puede potencialmente afectar a todo el proceso y, por extensión, a todos los threads dentro de ese

proceso. En contraste, un fallo en un proceso generalmente no afecta directamente a otros procesos en ejecución, proporcionando un mayor nivel de aislamiento y estabilidad del sistema [19].

En cuanto a la escalabilidad, los threads ofrecen ventajas significativas en sistemas con múltiples núcleos o procesadores. Múltiples threads de un proceso pueden ejecutarse verdaderamente en paralelo en diferentes núcleos, mientras que los procesos, aunque pueden distribuirse entre núcleos, tienen una granularidad más gruesa y mayores costos de comunicación [11].

Es importante señalar que algunos sistemas operativos modernos implementan modelos híbridos, como los «lightweight processes» o procesos ligeros, que combinan características de procesos y threads para optimizar el rendimiento y la flexibilidad [20].

Comprender estas diferencias es fundamental para los desarrolladores de software, ya que influye directamente en las decisiones de diseño relacionadas con la concurrencia, la paralelización y la arquitectura general de las aplicaciones.

6. Modelo de Concurrencia en Python y el Global Interpreter Lock

El modelo de concurrencia en Python presenta características únicas que lo distinguen de otros lenguajes de programación. Python implementa la concurrencia principalmente a través de su módulo `threading`, que permite la creación y gestión de hilos. Sin embargo, la ejecución de estos hilos está fuertemente influenciada por una característica fundamental de la implementación de referencia de Python (CPython): el Global Interpreter Lock (GIL) [5].

El GIL es un mecanismo de bloqueo que asegura que sólo un hilo ejecute código de bytes de Python en un momento dado. Esta restricción se implementó originalmente para simplificar el manejo de la memoria y garantizar la seguridad de los hilos en operaciones que no son atómicas en la implementación subyacente en C de Python [21].

El funcionamiento del GIL tiene implicaciones significativas en el rendimiento de programas Python multiproceso:

1. **Limitación del paralelismo real**: Aunque Python permite la creación de múltiples hilos, el GIL impide que estos se ejecuten verdaderamente en paralelo en múltiples núcleos de CPU para operaciones intensivas de CPU [22].

2. **Eficiencia en operaciones I/O**: El GIL se libera durante operaciones de I/O, permitiendo que otros hilos se ejecuten mientras un hilo está bloqueado en I/O. Esto hace que el «threading» en Python sea particularmente eficiente para aplicaciones limitadas por I/O [26].

3. **Impacto en el rendimiento**: En aplicaciones intensivas en CPU, el GIL puede llevar a una disminución del rendimiento en sistemas multinúcleo, ya que los hilos compiten por adquirir el GIL [5].

A pesar de estas limitaciones, Python ofrece varias alternativas para lograr concurrencia y paralelismo:

1. **Multiprocessing**: El módulo `multiprocessing` permite crear procesos separados, cada uno con su propio intérprete Python y GIL, permitiendo un verdadero paralelismo en sistemas multinúcleo [23].

2. **Asyncio**: Introducido en Python 3.4, este módulo proporciona un framework para programación asíncrona basada en corrutinas, ideal para aplicaciones con un alto grado de concurrencia en I/O [24].

3. **Implementaciones alternativas**: Algunas implementaciones de Python, como Jython o IronPython, no tienen GIL, permitiendo un verdadero multithreading [5].

Es importante destacar que, a pesar de las limitaciones impuestas por el GIL, el modelo de concurrencia de Python sigue siendo efectivo para muchas aplicaciones, especialmente aquellas limitadas por I/O. Además, el debate sobre el futuro del GIL en Python continúa, con propuestas y experimentos en curso para su posible eliminación en futuras versiones del lenguaje [25].

Comprender el modelo de concurrencia de Python y las implicaciones del GIL es crucial para los desarrolladores que buscan optimizar el rendimiento de aplicaciones Python en entornos concurrentes y paralelos. Este conocimiento permite tomar decisiones informadas sobre la arquitectura de la aplicación y la elección de las herramientas de concurrencia más apropiadas para cada caso de uso específico.

7. Implementación Básica de «Threading» en Python

Introducción al Módulo «threading»

El módulo «threading» es una parte fundamental de la biblioteca estándar de Python, diseñado para facilitar la implementación de programación concurrente basada en hilos. Este módulo proporciona una capa de abstracción de alto nivel sobre las primitivas de threading del sistema operativo, permitiendo a los desarrolladores crear y gestionar hilos de manera eficiente y portable [27].

El propósito principal del módulo «threading» es permitir que múltiples operaciones se ejecuten concurrentemente dentro de un mismo proceso de Python. Esto es particularmente útil para mejorar el rendimiento de aplicaciones que realizan operaciones de entrada/salida intensivas o que se benefician de la ejecución paralela en sistemas con múltiples núcleos de procesamiento [5].

Entre las funcionalidades clave que ofrece el módulo «threading» se encuentran:

1. Creación y gestión de hilos: Proporciona mecanismos para iniciar, pausar y detener hilos.

2. Sincronización: Ofrece primitivas como locks, semáforos y condiciones para coordinar la ejecución entre hilos.

3. Comunicación entre hilos: Facilita el intercambio seguro de datos entre hilos concurrentes.

4. Agrupación de hilos: Permite organizar y gestionar grupos de hilos relacionados.

5. Temporizadores: Ofrece la capacidad de ejecutar funciones en hilos separados después de un intervalo de tiempo especificado.

Es importante destacar que el módulo «threading» opera dentro de las limitaciones impuestas por el Global Interpreter Lock (GIL) de Python, lo que significa que, aunque puede mejorar significativamente el rendimiento en tareas limitadas por I/O, puede no proporcionar ganancias de rendimiento significativas para tareas intensivas en CPU en implementaciones estándar de Python como CPython [22].

El módulo «threading» se basa en gran medida en la API de «threading» de bajo nivel proporcionada por el módulo «thread», pero ofrece una interfaz más amigable y de alto nivel. Esto hace que sea más accesible para los desarrolladores, al tiempo que proporciona un mayor control sobre la gestión de hilos en comparación con otras alternativas como el módulo «concurrent.futures» [21].

En el contexto del desarrollo de software moderno, el módulo «threading» de Python juega un papel crucial en la implementación de aplicaciones concurrentes, desde servidores web y sistemas de procesamiento de datos hasta interfaces de usuario responsivas y sistemas de automatización. Su integración con otras bibliotecas y frameworks de Python lo convierte en una herramienta versátil para abordar una amplia gama de desafíos de programación concurrente [26].

8. Creación y Gestión de Threads Básicos en Python

La implementación de threads en Python utilizando el módulo «threading» es relativamente sencilla. A continuación, se presentan ejemplos de código que demuestran la creación y gestión básica de threads.

Creación de un Thread Básico

El siguiente ejemplo muestra cómo crear y ejecutar un thread básico:

```python
import threading
import time

def worker():
    print(f"Thread {threading.current_thread().name} iniciando")
    time.sleep(2)
    print(f"Thread {threading.current_thread().name} finalizando")

# Creación del thread
thread = threading.Thread(target=worker, name="WorkerThread")

# Inicio del thread
thread.start()

# Espera a que el thread termine
thread.join()

print("Programa principal finalizando")
```

En este ejemplo, se define una función `worker` que será ejecutada por el thread. Se crea un objeto `Thread`, especificando la función objetivo y un nombre para el thread. El método `start()` inicia la ejecución del thread, y `join()` espera a que el thread termine antes de continuar con la ejecución del programa principal [27].

Creación de Múltiples Threads

Para demostrar la ejecución concurrente, se pueden crear múltiples threads:

```
import threading
import time

def worker(id):
    print(f"Thread {id} iniciando")
    time.sleep(2)
    print(f"Thread {id} finalizando")

threads = []
for i in range(5):
    thread = threading.Thread(target=worker, args=(i,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Todos los threads han finalizado")
```

Este ejemplo crea cinco threads que se ejecutan concurrentemente. Cada thread ejecuta la función `worker` con un identificador único [5].

Uso de una Subclase de Thread

Alternativamente, se puede crear una subclase de `Thread` para encapsular la funcionalidad:

```
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, id):
        super().__init__()
        self.id = id

    def run(self):
        print(f"Thread {self.id} iniciando")
        time.sleep(2)
        print(f"Thread {self.id} finalizando")

threads = []
for i in range(5):
    thread = MyThread(i)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Todos los threads han finalizado")
```

En este enfoque, se define una clase `MyThread` que hereda de `threading.Thread`. El método `run()` se sobrescribe para definir el comportamiento del thread [22].

Estos ejemplos demuestran los conceptos básicos de creación y gestión de threads en Python. Es importante notar que, debido al Global Interpreter Lock (GIL), estos threads no se ejecutarán en paralelo verdadero para operaciones intensivas de CPU en CPython, pero pueden ser muy efectivos para tareas limitadas por I/O [21].

9. Métodos Principales de la Clase Thread

La clase Thread en Python proporciona varios métodos para controlar el ciclo de vida y el comportamiento de los hilos. Entre estos, tres métodos destacan por su importancia y uso frecuente: `start()`, `join()`, y `is_alive()`. A continuación, se explica en detalle cada uno de estos métodos.

Método start()

El método `start()` se utiliza para iniciar la ejecución de un hilo. Cuando se llama a este método, Python realiza las siguientes acciones:

1. Crea un nuevo hilo en el sistema operativo.
2. Invoca el método `run()` del objeto Thread en el nuevo hilo.

Es importante notar que `start()` debe ser llamado solo una vez por objeto Thread. Intentar llamarlo más de una vez resultará en una excepción `RuntimeError` [27].

Ejemplo de uso:

```
        thread = threading.Thread(target=worker_function)
        thread.start()  # Inicia la ejecución del hilo
```

El método `start()` retorna inmediatamente, permitiendo que el programa continúe su ejecución sin esperar a que el hilo complete su tarea.

Método join()

El método `join()` se utiliza para esperar a que un hilo complete su ejecución. Este método bloquea el hilo que lo llama hasta que el hilo sobre el que se invoca termine su ejecución o hasta que transcurra un tiempo de espera opcional.

Características principales de `join()`:

- Puede especificar un tiempo de espera máximo como argumento (en segundos).
- Si no se especifica un tiempo de espera, esperará indefinidamente.
- Retorna `None` si el hilo terminó dentro del tiempo de espera, o un booleano indicando si el hilo aún está vivo.

Ejemplo de uso:

```
thread.join()  # Espera indefinidamente
thread.join(timeout=5)  # Espera hasta 5 segundos
```

El uso de `join()` es crucial para sincronizar la ejecución entre hilos y asegurar que ciertos hilos hayan completado su tarea antes de continuar con el programa principal [5].

Método is_alive()

El método `is_alive()` se utiliza para verificar si un hilo está actualmente en ejecución. Retorna `True` si el hilo ha sido iniciado y no ha terminado aún, y `False` en caso contrario.

Este método es útil para:

- Verificar el estado de un hilo antes de realizar ciertas operaciones.
- Implementar lógica de espera o reintento basada en el estado de los hilos.
- Depurar y monitorear el comportamiento de los hilos en una aplicación.

Ejemplo de uso:

```
while thread.is_alive():
    print("El hilo aún está en ejecución")
    time.sleep(1)
```

Es importante tener en cuenta que el estado de un hilo puede cambiar inmediatamente después de llamar a `is_alive()`, por lo que su valor debe tratarse como una aproximación en escenarios de alta concurrencia [22].

La comprensión y el uso adecuado de estos métodos son fundamentales para el manejo efectivo de hilos en Python. Permiten un control preciso sobre el ciclo de vida de los hilos, facilitando la implementación de patrones de concurrencia robustos y eficientes.

10. Sincronización y Comunicación entre Threads

Desafíos de la Programación Concurrente

La programación concurrente, aunque poderosa, introduce una serie de desafíos significativos que los desarrolladores deben abordar para garantizar la corrección y eficiencia de sus programas. Dos de los problemas más críticos y frecuentes en la programación concurrente son las condiciones de carrera (race conditions) y los interbloqueos (deadlocks).

Condiciones de Carrera (Race Conditions)

Las condiciones de carrera ocurren cuando múltiples threads acceden y manipulan datos compartidos de manera concurrente, y el resultado final depende del orden de ejecución de los threads [17]. Este fenómeno puede llevar a comportamientos impredecibles y errores difíciles de reproducir y depurar.

Considere el siguiente ejemplo conceptual:

```
1 contador = 0
2
3 def incrementar():
4     global contador
5     temp = contador
6     temp = temp + 1
7     contador = temp
8
9 # Suponga que dos threads ejecutan esta función concurrentemente
```

En este escenario, si dos threads ejecutan la función `incrementar()` simultáneamente, podrían leer el mismo valor inicial, incrementarlo y escribirlo de vuelta, resultando en un incremento único en lugar de dos. Este tipo de errores sutiles pueden tener consecuencias graves en sistemas de producción.

Las condiciones de carrera pueden manifestarse de diversas formas, incluyendo:

- Pérdida de actualizaciones
- Lecturas sucias (cuando un thread lee datos parcialmente actualizados por otro thread)
- Inconsistencias en estructuras de datos complejas

La prevención de condiciones de carrera generalmente implica el uso de mecanismos de sincronización para garantizar el acceso exclusivo a los recursos compartidos, un tema que se abordará en detalle más adelante [11].

Interbloqueos (Deadlocks)

Los interbloqueos representan otro desafío crítico en la programación concurrente. Un interbloqueo ocurre cuando dos o más threads se bloquean mutuamente, cada uno esperando que el otro libere un recurso, resultando en un estancamiento permanente [15].

Para que se produzca un interbloqueo, deben cumplirse cuatro condiciones (conocidas como las condiciones de Coffman):

1. Exclusión mutua: Al menos un recurso debe ser retenido en modo no compartible.
2. Retención y espera: Un thread debe retener al menos un recurso mientras espera adquirir recursos adicionales retenidos por otros threads.
3. No apropiación: Los recursos no pueden ser forzosamente quitados de un thread; deben ser liberados voluntariamente.
4. Espera circular: Debe existir un conjunto circular de threads, cada uno esperando un recurso retenido por el siguiente thread en el ciclo.

Un ejemplo clásico de escenario propenso a interbloqueos es el «problema de la cena de los filósofos», donde múltiples filósofos («threads») compiten por un número limitado de tenedores (recursos compartidos) [28].

La prevención y detección de interbloqueos son áreas críticas en el diseño de sistemas concurrentes. Las estrategias para abordar los interbloqueos incluyen:

- Prevención: Diseñar el sistema para que al menos una de las condiciones de Coffman no pueda ocurrir.
- Evasión: Tomar decisiones dinámicas sobre la asignación de recursos para evitar estados inseguros.

- Detección y recuperación: Permitir que los interbloqueos ocurran, pero implementar mecanismos para detectarlos y recuperarse de ellos.

Estos desafíos subrayan la complejidad inherente a la programación concurrente y la necesidad de un diseño cuidadoso y una implementación rigurosa al trabajar con threads. La comprensión profunda de estos problemas es crucial para desarrollar sistemas concurrentes robustos y fiables.

11. Herramientas de Sincronización

Locks (Cerrojos)

Los locks, también conocidos como cerrojos o mutex (exclusión mutua), son una de las herramientas fundamentales de sincronización en la programación concurrente. Su función principal es proteger recursos compartidos, asegurando que solo un thread a la vez pueda acceder a un recurso crítico [8].

Funcionamiento de los Locks

Un lock opera como un mecanismo de bloqueo binario que puede estar en uno de dos estados: bloqueado o desbloqueado. Cuando un thread adquiere un lock, este pasa al estado bloqueado, impidiendo que otros threads lo adquieran hasta que sea liberado. Los locks proporcionan las siguientes operaciones básicas:

- Adquirir (lock): Intenta obtener el lock. Si está disponible, lo adquiere y continúa la ejecución. Si no está disponible, el thread se bloquea hasta que pueda adquirirlo.
- Liberar (unlock): Libera el lock, permitiendo que otros threads lo adquieran.

Implementación en Python

En Python, los locks se implementan a través de la clase `threading.Lock`. He aquí un ejemplo básico de su uso:

```python
import threading

# Recurso compartido
counter = 0
# Creación del lock
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

# Creación de threads
threads = [threading.Thread(target=increment) for _ in range(5)]

# Inicio de threads
for thread in threads:
    thread.start()

# Espera a que todos los threads terminen
```

```
22 for thread in threads:
23     thread.join()
24
25 print(f"Contador final: {counter}")
```

En este ejemplo, el lock protege la operación de incremento del contador, asegurando que solo un thread a la vez pueda modificar el valor [22].

Ventajas de los Locks

- Simplicidad: Los locks son conceptualmente simples y fáciles de usar.
- Eficiencia: Proporcionan una sobrecarga mínima en comparación con otras herramientas de sincronización más complejas.
- Prevención de condiciones de carrera: Garantizan el acceso exclusivo a recursos compartidos.

Desafíos y Consideraciones

A pesar de su utilidad, los locks presentan algunos desafíos:

- Deadlocks: Si no se gestionan adecuadamente, pueden llevar a situaciones de interbloqueo.
- Granularidad: Un lock demasiado amplio puede reducir el paralelismo, mientras que uno demasiado fino puede aumentar la complejidad y la sobrecarga.
- Inversión de prioridad: Puede ocurrir cuando un thread de baja prioridad mantiene un lock necesario para un thread de alta prioridad.

Variantes de Locks

Python también ofrece variantes más especializadas de locks:

- `RLock` (Reentrant Lock): Permite que un mismo thread adquiera el lock múltiples veces sin bloquearse a sí mismo.
- `Condition`: Combina la funcionalidad de un lock con la capacidad de esperar por una condición específica.

El uso efectivo de locks requiere una comprensión profunda de la arquitectura del sistema y los patrones de acceso a los recursos compartidos. Cuando se implementan correctamente, los locks son una herramienta poderosa para garantizar la consistencia y prevenir condiciones de carrera en programas concurrentes [21].

Semáforos

Los semáforos son otra herramienta fundamental de sincronización en la programación concurrente. Introducidos por Edsger Dijkstra en 1965, los semáforos proporcionan un mecanismo más flexible que los locks para controlar el acceso a recursos compartidos y coordinar la ejecución de múltiples threads [29].

Concepto y Funcionamiento

Un semáforo es esencialmente un contador que puede ser decrementado o incrementado por threads concurrentes. El valor del semáforo representa el número de unidades de un recurso disponible. Los semáforos soportan dos operaciones principales:

- `acquire()` (también conocido como `wait()` o `P()`): Decrementa el contador. Si el contador es cero, el thread se bloquea hasta que el contador sea mayor que cero.
- `release()` (también conocido como `signal()` o `V()`): Incrementa el contador y desbloquea un thread en espera, si lo hay.

Tipos de Semáforos

Existen dos tipos principales de semáforos:

1. **Semáforos binarios**: Solo pueden tomar los valores 0 o 1. Son similares a los locks, pero permiten que un thread libere un semáforo que fue adquirido por otro thread.
2. **Semáforos de conteo**: Pueden tomar cualquier valor entero no negativo. Son útiles para representar recursos con múltiples instancias.

Implementación en Python

En Python, los semáforos se implementan a través de la clase `threading.Semaphore`. Aquí hay un ejemplo de su uso:

```python
import threading
import time

# Creación de un semáforo con valor inicial 2
semaphore = threading.Semaphore(2)

def worker(id):
    print(f"Thread {id} intentando adquirir el semáforo")
    with semaphore:
        print(f"Thread {id} adquirió el semáforo")
        time.sleep(2)  # Simula algún trabajo
    print(f"Thread {id} liberó el semáforo")

# Creación y ejecución de 5 threads
threads = [threading.Thread(target=worker, args=(i,)) for i in range(5)]
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()
```

En este ejemplo, el semáforo limita el número de threads que pueden ejecutar la sección crítica simultáneamente a dos [22].

Aplicaciones de los Semáforos

Los semáforos son particularmente útiles en varios escenarios:

- **Control de acceso a recursos limitados**: Por ejemplo, controlar el número de conexiones simultáneas a una base de datos.
- **Sincronización de productor-consumidor**: Coordinar threads productores y consumidores que operan sobre un buffer compartido.
- **Implementación de barreras**: Sincronizar múltiples threads para que esperen en un punto específico hasta que todos hayan llegado.

Ventajas de los Semáforos

- **Flexibilidad**: Pueden modelar una variedad de problemas de sincronización más allá de la exclusión mutua simple.
- **Eficiencia**: Proporcionan una forma eficiente de gestionar múltiples instancias de un recurso.
- **Prevención de condiciones de carrera**: Al igual que los locks, ayudan a prevenir condiciones de carrera en el acceso a recursos compartidos.

Desafíos y Consideraciones

El uso de semáforos también presenta algunos desafíos:

- **Complejidad**: Pueden ser más difíciles de razonar y depurar que los locks simples, especialmente en sistemas complejos.
- **Riesgo de deadlock**: El uso incorrecto de semáforos puede llevar a situaciones de interbloqueo.
- **Inversión de prioridad**: Al igual que con los locks, puede ocurrir inversión de prioridad si no se maneja adecuadamente.

Los semáforos son una herramienta poderosa en el arsenal de sincronización del programador concurrente. Su versatilidad los hace adecuados para una amplia gama de problemas de sincronización, pero requieren una comprensión cuidadosa y una implementación precisa para evitar errores sutiles [17].

Eventos

Los eventos son otra herramienta importante en el arsenal de sincronización para la programación concurrente. Proporcionan un mecanismo simple pero poderoso para la comunicación entre threads, permitiendo que un thread señalice a otros threads sobre la ocurrencia de una condición o estado específico [8].

Concepto y Funcionamiento

Un evento es esencialmente un objeto que actúa como una señal entre threads. Tiene dos estados principales: establecido (set) o no establecido (unset). Los threads pueden esperar a que un evento se establezca, o pueden establecer el evento para notificar a otros threads. Las operaciones principales de un evento son:

- `wait()`: Bloquea el thread hasta que el evento se establezca.
- `set()`: Establece el evento, despertando a todos los threads que están esperando.
- `clear()`: Restablece el evento a su estado no establecido.
- `is_set()`: Comprueba si el evento está establecido.

Implementación en Python

En Python, los eventos se implementan a través de la clase `threading.Event`. Aquí hay un ejemplo de su uso:

```python
import threading
import time

def waiter(event, name):
    print(f"{name} esperando el evento")
    event.wait()
    print(f"{name} recibió el evento")

def setter(event, name, delay):
    print(f"{name} esperando {delay} segundos antes de establecer el evento")
    time.sleep(delay)
    event.set()
    print(f"{name} estableció el evento")

# Creación del evento
event = threading.Event()

# Creación de threads
threads = [
    threading.Thread(target=waiter, args=(event, "Waiter 1")),
    threading.Thread(target=waiter, args=(event, "Waiter 2")),
    threading.Thread(target=setter, args=(event, "Setter", 3))
]

# Inicio de threads
for thread in threads:
    thread.start()

# Espera a que todos los threads terminen
for thread in threads:
    thread.join()
```

En este ejemplo, dos threads esperan a que el evento se establezca, mientras que un tercer thread lo establece después de una espera de 3 segundos [22].

Aplicaciones de los Eventos

Los eventos son particularmente útiles en varios escenarios:

- **Señalización de inicio**: Sincronizar el inicio de múltiples threads.
- **Notificación de finalización**: Indicar a otros threads que una tarea ha sido completada.
- **Control de flujo**: Pausar y reanudar la ejecución de threads basándose en ciertas condiciones.

- **Implementación de timeouts**: Combinar eventos con temporizadores para implementar esperas con límite de tiempo.

Ventajas de los Eventos

- **Simplicidad**: Los eventos proporcionan una interfaz simple y fácil de entender para la sincronización.
- **Eficiencia**: Son eficientes para escenarios de «esperar y notificar» con múltiples oyentes.
- **Flexibilidad**: Pueden ser utilizados en una variedad de patrones de sincronización.

Consideraciones y Mejores Prácticas

Al trabajar con eventos, es importante tener en cuenta:

- **Granularidad**: Usar eventos para sincronización de grano grueso, no para proteger accesos a recursos compartidos (para eso son más apropiados los locks).
- **Orden de ejecución**: Los eventos no garantizan un orden específico en el que los threads esperando serán notificados.
- **Condiciones de carrera**: Tener cuidado con las condiciones de carrera entre la comprobación del estado del evento y la espera.

Variantes y Extensiones

Algunas implementaciones de eventos ofrecen características adicionales:

- **Eventos con auto-reset**: Se restablecen automáticamente después de despertar a un thread.
- **Eventos con conteo**: Combinan características de eventos y semáforos, permitiendo un número específico de threads para proceder.

Los eventos son una herramienta valiosa en el diseño de sistemas concurrentes, proporcionando un mecanismo sencillo pero potente para la coordinación entre threads. Su uso efectivo puede simplificar significativamente la lógica de sincronización en muchos escenarios de programación concurrente [21].

Colas

Las colas son estructuras de datos fundamentales en la programación concurrente que facilitan la comunicación y sincronización entre threads. Proporcionan un mecanismo seguro para threads para intercambiar datos y coordinar su ejecución, especialmente útil en escenarios de productor-consumidor [8].

Concepto y Funcionamiento

Una cola en el contexto de la programación concurrente es una estructura de datos que permite a los threads añadir elementos (enqueue) y retirar elementos (dequeue) de manera segura para threads. Las operaciones principales son:

- `put()`: Añade un elemento a la cola. Si la cola está llena, el thread se bloquea hasta que haya espacio disponible.
- `get()`: Retira y devuelve un elemento de la cola. Si la cola está vacía, el thread se bloquea hasta que haya un elemento disponible.
- `task_done()`: Indica que una tarea ha sido completada.
- `join()`: Bloquea hasta que todos los elementos en la cola hayan sido procesados.

Implementación en Python

En Python, las colas thread-safe se implementan a través del módulo `queue`. Aquí hay un ejemplo de su uso:

```python
import threading
import queue
import time

def producer(q):
    for i in range(5):
        item = f"item-{i}"
        q.put(item)
        print(f"Producido: {item}")
        time.sleep(1)

def consumer(q):
    while True:
        item = q.get()
        if item is None:
        break
        print(f"Consumido: {item}")
        q.task_done()

# Creación de la cola
q = queue.Queue()

# Creación de threads
producer_thread = threading.Thread(target=producer, args=(q,))
consumer_thread = threading.Thread(target=consumer, args=(q,))

# Inicio de threads
producer_thread.start()
consumer_thread.start()

# Espera a que el productor termine
producer_thread.join()

# Señal de finalización al consumidor
q.put(None)

# Espera a que el consumidor termine
consumer_thread.join()
```

En este ejemplo, un thread productor genera elementos y los coloca en la cola, mientras que un thread consumidor los retira y procesa [22].

Tipos de Colas en Python

Python ofrece varios tipos de colas en el módulo `queue`:

- `Queue`: Cola FIFO (First-In-First-Out) estándar.
- `LifoQueue`: Cola LIFO (Last-In-First-Out) o pila.
- `PriorityQueue`: Cola donde los elementos se ordenan por prioridad.

Aplicaciones de las Colas

Las colas son particularmente útiles en varios escenarios:

- **Patrón productor-consumidor**: Ideal para situaciones donde unos threads producen datos y otros los consumen.
- **Balanceo de carga**: Distribuir tareas entre múltiples threads trabajadores.
- **Buffering**: Manejar flujos de datos asincrónicos o de velocidad variable.
- **Desacoplamiento de componentes**: Permitir que diferentes partes de un programa operen de manera independiente.

Ventajas de las Colas

- **Thread-safety**: Las operaciones de cola son atómicas y seguras para threads.
- **Bloqueo automático**: Manejan automáticamente el bloqueo y desbloqueo de threads.
- **Flexibilidad**: Pueden adaptarse a diversos patrones de comunicación entre threads.
- **Escalabilidad**: Facilitan la implementación de sistemas escalables con múltiples productores y consumidores.

Consideraciones y Mejores Prácticas

Al trabajar con colas, es importante tener en cuenta:

- **Tamaño de la cola**: Considerar limitar el tamaño de la cola para evitar el consumo excesivo de memoria.
- **Manejo de excepciones**: Implementar un manejo adecuado de excepciones, especialmente para operaciones que puedan bloquearse.
- **Señalización de finalización**: Utilizar un valor centinela (como `None`) para señalar el fin de la producción.
- **Monitoreo**: Considerar implementar mecanismos para monitorear el tamaño y el rendimiento de la cola.

Las colas proporcionan un mecanismo robusto y flexible para la comunicación entre threads, simplificando significativamente la implementación de patrones comunes en programación concurrente. Su uso efectivo puede llevar a diseños más limpios y eficientes en sistemas «multithreading» complejos [21].

12. Ejemplos Prácticos de Sincronización

Para ilustrar el uso práctico de las herramientas de sincronización discutidas, se presentan una serie de ejemplos que abordan problemas comunes en programación concurrente.

Ejemplo 1: Protección de Recursos Compartidos con Locks

Este ejemplo muestra cómo utilizar un lock para proteger un contador compartido:

```
import threading

class SharedCounter:
    def __init__(self):
        self.count = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:
        self.count += 1

    def get_count(self):
        with self.lock:
        return self.count

def worker(counter, num_increments):
    for _ in range(num_increments):
        counter.increment()

counter = SharedCounter()
threads = []
for _ in range(5):
    t = threading.Thread(target=worker, args=(counter, 10000))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Conteo final: {counter.get_count()}")
```

Este ejemplo demuestra cómo un lock puede prevenir condiciones de carrera al asegurar que solo un thread a la vez pueda modificar el contador [8].

Ejemplo 2: Control de Acceso con Semáforos

Este ejemplo utiliza un semáforo para limitar el número de threads que pueden acceder a un recurso simultáneamente:

```
import threading
import time

class LimitedResource:
    def __init__(self, max_workers):
```

```
        self.semaphore = threading.Semaphore(max_workers)

    def use_resource(self, worker_id):
        with self.semaphore:
            print(f"Worker {worker_id} está usando el recurso")
            time.sleep(1)  # Simula el uso del recurso
            print(f"Worker {worker_id} ha terminado de usar el recurso")

def worker(resource, worker_id):
    for _ in range(3):
        resource.use_resource(worker_id)

resource = LimitedResource(max_workers=3)
threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(resource, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Este ejemplo muestra cómo un semáforo puede controlar el acceso a un recurso limitado, permitiendo que solo un número específico de threads lo utilicen simultáneamente [22].

Ejemplo 3: Sincronización de Inicio con Eventos

Este ejemplo utiliza un evento para sincronizar el inicio de múltiples threads:

```
import threading
import time
import random

def worker(start_event, worker_id):
    print(f"Worker {worker_id} está listo y esperando")
    start_event.wait()
    print(f"Worker {worker_id} ha comenzado")
    time.sleep(random.random())
    print(f"Worker {worker_id} ha terminado")

start_event = threading.Event()
threads = []

for i in range(5):
    t = threading.Thread(target=worker, args=(start_event, i))
    threads.append(t)
    t.start()

print("Preparando para iniciar los workers...")
time.sleep(2)
print("¡Iniciando todos los workers!")
start_event.set()

for t in threads:
    t.join()
```

Este ejemplo demuestra cómo un evento puede ser utilizado para sincronizar el inicio de múltiples threads, asegurando que todos comiencen su tarea principal al mismo tiempo [21].

Ejemplo 4: Patrón Productor-Consumidor con Colas

El siguiente ejemplo implementa el patrón productor-consumidor utilizando una cola. Este patrón es útil en la programación concurrente y se utiliza para coordinar la comunicación entre threads que producen datos (productores) y threads que procesan esos datos (consumidores).

```
1  import threading
2  import queue
3  import time
4  import random
5
6  def producer(q, num_items):
7      for i in range(num_items):
8          item = f"Item {i}"
9          q.put(item)
10         print(f"Producido: {item}")
11         time.sleep(random.random())
12     q.put(None)  # Señal de finalización
13
14 def consumer(q):
15     while True:
16         item = q.get()
17         if item is None:
18         break
19         print(f"Consumido: {item}")
20         time.sleep(random.random() * 2)
21         q.task_done()
22
23 q = queue.Queue(maxsize=5)
24 producer_thread = threading.Thread(target=producer, args=(q, 10))
25 consumer_thread = threading.Thread(target=consumer, args=(q,))
26
27 producer_thread.start()
28 consumer_thread.start()
29
30 producer_thread.join()
31 consumer_thread.join()
32
33 print("Producción y consumo completados")
```

1. **Importaciones (líneas 1-4):**
   - `threading`: Para crear y gestionar threads.
   - `queue`: Proporciona la implementación thread-safe de una cola.
   - `time`: Para simular el tiempo de procesamiento.
   - `random`: Para generar tiempos de espera aleatorios.
2. **Función productor (líneas 6-12):**
   - Recibe como argumentos la cola `q` y el número de items a producir `num_items`.
   - Genera items y los coloca en la cola usando `q.put(item)`.
   - Simula tiempo de producción variable con `time.sleep(random.random())`.
   - Al finalizar, envía una señal de terminación (`None`) a la cola.
3. **Función consumidor (líneas 14-21):**
   - Ejecuta un bucle infinito que obtiene items de la cola con `q.get()`.

- Si el item es `None`, termina el bucle (señal de finalización).
- Simula el tiempo de procesamiento con `time.sleep(random.random() * 2)`.
- Llama a `q.task_done()` para indicar que ha terminado de procesar el item.

4. **Creación de la cola (línea 23):**
   - Se crea una cola con un tamaño máximo de 5 items.
   - Esto introduce un límite en la producción, haciendo que el productor espere si la cola está llena.
5. **Creación de threads (líneas 24-25):**
   - Se crean dos threads: uno para el productor y otro para el consumidor.
   - El thread productor se configura para producir 10 items.
6. **Inicio de threads (líneas 27-28):**
   - Se inician ambos threads con el método `start()`.
7. **Espera de finalización (líneas 30-31):**
   - Se usa `join()` para esperar a que ambos threads terminen su ejecución.

Funcionamiento del Patrón

- El productor genera items y los coloca en la cola. Si la cola está llena (5 items), el productor se bloquea hasta que haya espacio disponible.
- El consumidor continuamente intenta obtener items de la cola. Si la cola está vacía, se bloquea hasta que haya un item disponible.
- El uso de `queue.Queue` garantiza la sincronización thread-safe entre el productor y el consumidor.
- Los tiempos de espera aleatorios simulan variaciones en los tiempos de producción y consumo, demostrando cómo la cola actúa como un buffer entre productor y consumidor.
- La señal de finalización (`None`) permite que el consumidor sepa cuándo terminar su ejecución.

Ventajas del Patrón

- **Desacoplamiento:** El productor y el consumidor no necesitan conocerse mutuamente, solo interactúan a través de la cola.
- **Control de flujo:** La cola actúa como un buffer, permitiendo que productor y consumidor trabajen a ritmos diferentes.
- **Sincronización implícita:** La implementación thread-safe de `queue.Queue` maneja automáticamente los problemas de sincronización.

Estos ejemplos prácticos ilustran cómo las diferentes herramientas de sincronización pueden ser aplicadas para resolver problemas comunes en programación concurrente. Cada herramienta tiene sus propias fortalezas y es más adecuada para ciertos escenarios. La elección de la herramienta correcta depende de las necesidades específicas del problema y de las características del sistema concurrente que se está desarrollando.

13. Optimización de Programas con «Threading»

Situaciones en las que el «Threading» Mejora el Rendimiento

El «threading» puede mejorar significativamente el rendimiento de los programas en ciertas situaciones. Es crucial entender estos escenarios para aprovechar eficazmente la programación concurrente. A continuación, se analizan las principales situaciones en las que el «threading» puede ofrecer beneficios de rendimiento notables.

Operaciones Intensivas en E/S

Una de las situaciones más comunes donde el «threading» mejora el rendimiento es en programas con operaciones intensivas en entrada/salida (E/S). Estas operaciones incluyen:

- Lectura y escritura de archivos
- Comunicaciones de red
- Consultas a bases de datos
- Llamadas a APIs externas

En estos casos, un thread puede iniciar una operación de E/S y luego ceder el control, permitiendo que otros threads se ejecuten mientras se espera la finalización de la E/S. Esto maximiza la utilización del CPU y reduce el tiempo total de ejecución [8].

Ejemplo conceptual:

```
import threading
import requests

def fetch_url(url):
    response = requests.get(url)
    print(f"Fetched: {url}, Status: {response.status_code}")

urls = ["http://example.com", "http://example.org", "http://example.net"]
threads = []

for url in urls:
    thread = threading.Thread(target=fetch_url, args=(url,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

En este ejemplo, múltiples solicitudes HTTP se realizan concurrentemente, mejorando significativamente el tiempo total de ejecución en comparación con un enfoque secuencial.

Procesamiento Paralelo de Datos

Para tareas que involucran el procesamiento de grandes cantidades de datos independientes, el «threading» puede distribuir el trabajo entre múltiples núcleos de CPU, acelerando significativamente el procesamiento [22].

Escenarios típicos incluyen:

- Procesamiento de imágenes o video
- Análisis de grandes conjuntos de datos
- Simulaciones científicas
- Operaciones matriciales y vectoriales

Ejemplo conceptual:

```
import threading
import numpy as np

def process_chunk(chunk):
    # Simula algún procesamiento intensivo
    result = np.mean(chunk ** 2)
    print(f"Processed chunk, result: {result}")

data = np.random.rand(1000000)
chunk_size = len(data) // 4
threads = []

for i in range(4):
    start = i * chunk_size
    end = start + chunk_size
    chunk = data[start:end]
    thread = threading.Thread(target=process_chunk, args=(chunk,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Este ejemplo divide un gran array en chunks y procesa cada chunk en un thread separado, aprovechando múltiples núcleos de CPU.

Aplicaciones de Interfaz de Usuario

En aplicaciones con interfaces de usuario gráficas (GUI), el «threading» puede mejorar significativamente la capacidad de respuesta. Permite que las operaciones de larga duración se ejecuten en threads separados, manteniendo la interfaz de usuario receptiva [26].

Beneficios clave:

- Prevención de «congelamiento» de la interfaz
- Actualización suave de la interfaz durante operaciones largas
- Mejor experiencia de usuario en general

Servidores y Aplicaciones de Red

Los servidores y aplicaciones de red se benefician enormemente del «threading», permitiendo manejar múltiples conexiones simultáneamente. Esto es crucial para:

- Servidores web
- Servidores de bases de datos
- Aplicaciones de chat o mensajería
- Sistemas de monitoreo en tiempo real

El «threading» permite que cada conexión o solicitud de cliente sea manejada en un thread separado, maximizando el throughput y la capacidad de respuesta del servidor [21].

Cálculos Asíncronos y Tareas en Segundo Plano

El «threading» es ideal para ejecutar cálculos asíncronos o tareas en segundo plano que no requieren interacción inmediata con el usuario. Ejemplos incluyen:

- Generación de informes
- Cálculos predictivos
- Actualización de cachés
- Tareas de mantenimiento programadas

Estas tareas pueden ejecutarse en threads separados, permitiendo que el programa principal continúe su ejecución sin interrupciones.

14. Visor del Historial del Portapapeles: Una Herramienta para Mejorar la Productividad

El Visor del Historial del Portapapeles es una aplicación diseñada para mejorar significativamente la eficiencia y comodidad en la gestión de información copiada en su computadora. Esta herramienta ofrece una solución práctica para uno de los desafíos comunes en el trabajo diario con ordenadores: la necesidad de acceder rápidamente a múltiples elementos copiados previamente.

Funcionalidad Principal

- **Historial de Copiado**: La aplicación mantiene un registro de los últimos 20 elementos copiados al portapapeles de su sistema. Esto significa que ya no tendrá que preocuparse por perder información importante que copió hace unos momentos.

- **Interfaz Visual Intuitiva**: Todos los elementos copiados se muestran en una lista clara y fácil de navegar. Cada entrada muestra una vista previa del contenido copiado, permitiéndole identificar rápidamente el elemento que necesita.

- **Acceso Rápido**: Con un simple movimiento del mouse, puede acceder a cualquier elemento de su historial de copiado. No más copiar y pegar repetitivo o búsqueda en documentos para encontrar información que ya había copiado.

- **Pegado Automático**: Al mantener el cursor sobre un elemento del historial durante un segundo, el contenido se copia automáticamente a su portapapeles actual y se pega en la aplicación activa. Esto agiliza enormemente el proceso de recuperar y utilizar información previamente copiada.

- **Monitoreo Continuo**: La aplicación funciona silenciosamente en segundo plano, capturando cada elemento que copia sin interrumpir su flujo de trabajo.

Beneficios para el Usuario

- **Ahorro de Tiempo**: Recupere rápidamente información que copió anteriormente sin tener que buscar en documentos o navegar por ventanas.

- **Mejora de la Productividad**: Cambie fácilmente entre múltiples fragmentos de información copiada, ideal para tareas que requieren compilar datos de varias fuentes.

- **Reducción de Errores**: Evite errores comunes como sobrescribir accidentalmente información importante en el portapapeles o olvidar contenido crítico que fue copiado anteriormente.

- **Flujo de Trabajo Ininterrumpido**: La naturaleza no intrusiva de la aplicación significa que puede beneficiarse de sus características sin alterar sus hábitos de trabajo establecidos.

- **Versatilidad**: Útil para una amplia gama de tareas, desde redacción y programación hasta investigación y gestión de datos.

Cómo Utilizar la Aplicación

1. **Inicio**: Al abrir la aplicación, verá una ventana que muestra su historial de portapapeles.

2. **Copiado Normal**: Continúe copiando información como lo hace habitualmente (Ctrl+C o clic derecho y copiar). La aplicación registrará automáticamente cada elemento copiado.

3. **Visualización del Historial**: En cualquier momento, puede abrir la ventana de la aplicación para ver una lista de sus elementos copiados recientemente.

4. **Recuperación de Elementos**: Para reutilizar un elemento copiado anteriormente, simplemente mueva el cursor sobre él en la lista y manténgalo allí por un segundo. El elemento se copiará automáticamente y se pegará en su aplicación activa.

5. **Personalización**: Puede ajustar el tamaño de la fuente para una mejor visibilidad y borrar el historial cuando lo desee.

El Visor del Historial del Portapapeles es una herramienta útil para cualquier persona que trabaje intensivamente con procedimientos rutinarios de copiar y pegar información, tales como el llenado de un formulario entre distintas aplicaciones. Al proporcionar un acceso rápido y fácil a su historial de copiado, esta aplicación no solo ahorra tiempo, sino que también mejora la precisión y eficiencia en una amplia gama de tareas informáticas. Ya sea que esté redactando un informe, programando, investigando o simplemente gestionando información diaria, esta herramienta se integrará perfectamente en su flujo de trabajo, mejorando su productividad sin esfuerzo adicional.

15. Análisis del Script pp.py

El script `pp.py` proporciona una implementación práctica de varios conceptos de «threading» y sincronización que se han discutido anteriormente. A continuación, se analizan las partes clave del script.

Importación de Módulos y Configuración Inicial

```
import tkinter as tk
from tkinter import ttk
import win32clipboard
import win32con
import pyautogui
import threading
import time
from collections import deque
from pynput import keyboard
```

Esta sección importa los módulos necesarios, incluyendo «threading», que es necesaria para la implementación de concurrencia en Python, como se discutió en la sección «Implementación Básica de “Threading” en Python».

Definición de la Clase Principal

```
class ClipboardHistoryViewer:
    def __init__(self, root):
        self.root = root
        self.root.title("Visor del Historial del Portapapeles")
        self.clipboard_history = deque(maxlen=20)
        self.font_size = 10
        self.hover_item = None
        self.hover_time = 0
        self.last_copied_content = None
        self.clipboard_lock = threading.Lock()
        self.setup_ui()
        self.start_monitoring()
        self.start_keyboard_listener()
```

La clase `ClipboardHistoryViewer` utiliza un `threading.Lock()` (línea 20) para sincronización, lo cual se relaciona con nuestra discusión sobre «Locks (cerrojos)» en la sección de herramientas de sincronización (Sección 11).

Monitoreo del Portapapeles

```
74 def start_monitoring(self):
75     self.monitoring_thread = threading.Thread(target=self.monitor_clipboard,
       ↪ daemon=True)
76     self.monitoring_thread.start()
77 
78 def monitor_clipboard(self):
79     last_content = ""
80     while True:
81         try:
82             with self.clipboard_lock:
83                 content = self.get_clipboard_content()
84             if content != last_content:
85                 self.root.after(0, self.add_to_history, content)
86                 last_content = content
87         except Exception as e:
88             print(f"Error al monitorear el portapapeles: {e}")
89         time.sleep(0.5)
```

Esta parte del script implementa un thread separado para monitorear el portapapeles (líneas 75-76), lo cual es un ejemplo práctico de «Creación y Gestión de Threads Básicos en Python» dicutidas en la Sección 8. El uso de `with self.clipboard_lock:` (línea 82) demuestra la aplicación práctica de locks para proteger recursos compartidos, como se explicó en «Ejemplos Prácticos de Sincronización» (Sección 12).

Manejo de Eventos de Teclado

```
136     def start_keyboard_listener(self):
137         def on_press(key):
138             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
139                 self.ctrl_pressed = True
140             elif hasattr(key, 'char') and key.char == 'c' and self.ctrl_pressed
                ↪ :
141                 self.root.after(100, self.update_on_ctrl_c)
142 
143         def on_release(key):
144             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
145                 self.ctrl_pressed = False
146 
147         self.ctrl_pressed = False
148         listener = keyboard.Listener(on_press=on_press, on_release=on_release)
149         listener.start()
```

Esta sección implementa un listener de teclado en un thread separado (línea 148), lo cual es otro ejemplo de la aplicación de «threading» para mejorar la capacidad de respuesta de la aplicación, como se discutió en «Optimización de Programas con “Threading” mejora el rendimiento» (Sección 13).

Actualización de la Interfaz de Usuario

```
151 def update_on_ctrl_c(self):
152     with self.clipboard_lock:
```

```
153         content = self.get_clipboard_content()
154         self.add_to_history(content)
```

Este método utiliza un lock para asegurar el acceso seguro al portapapeles (línea 152), demostrando nuevamente el uso práctico de locks para la sincronización, como se explicó en la sección de herramientas de sincronización (Sección 11).

Manejo de Eventos de la Interfaz de Usuario

```
167     def on_mouse_over(self, event):
168     item = self.tree.identify_row(event.y)
169     if item != self.hover_item:
170         self.hover_item = item
171         self.hover_time = time.time()
172     elif item and time.time() - self.hover_time >= 1:
173         content = self.tree.item(item, "values")[1]
174         if content.endswith("..."):
175             content = list(self.clipboard_history)[self.tree.index(item)]
176 
177         if content != self.last_copied_content:
178             with self.clipboard_lock:
179                 self.set_clipboard_content(content)
180             self.root.update()
181             pyautogui.hotkey('ctrl', 'v')
182             self.last_copied_content = content
183             self.status_var.set(f"Copiado: {content[:30]}...")
184             self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
                    ↪ green_circle',))
185             self.root.after(2000, lambda: self.reset_status_and_color(item))
186         else:
187             self.status_var.set("Contenido ya copiado.")
188             self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
                    ↪ green_circle',))
189             self.root.after(2000, lambda: self.reset_status_and_color(item))
```

Este método maneja los eventos del mouse en la interfaz de usuario. El uso de `with self.clipboard_lock:` (línea 178) para proteger el acceso al portapapeles es otro ejemplo de la aplicación de locks para evitar condiciones de carrera, como se discutió en «Sincronización y Comunicación entre "Threads"» (Sección 10).

En conjunto, este script demuestra la aplicación práctica de varios conceptos de «threading» y sincronización que se han discutido a lo largo de este artículo. Utiliza threads para mejorar la capacidad de respuesta de la aplicación, ejemplificando así el empleo de locks para proteger recursos compartidos, el manejo de eventos de usuario, todo lo cual se alinea con las mejores prácticas de programación concurrente en Python.

A continuación se presenta el script completo, que se puede descargar del repositorio github en el que se incluyen las instrucciones para descargar las librerías necesarias para su ejecución: `https://github.com/JavierSalasGarcia/ClipboardPlus`

```
1  import tkinter as tk
2  from tkinter import ttk
3  import win32clipboard
4  import win32con
5  import pyautogui
6  import threading
7  import time
8  from collections import deque
9  from pynput import keyboard
10 
11 class ClipboardHistoryViewer:
```

```
12      def __init__(self, root):
13          self.root = root
14          self.root.title("Visor del Historial del Portapapeles")
15          self.clipboard_history = deque(maxlen=20)
16          self.font_size = 10
17          self.hover_item = None
18          self.hover_time = 0
19          self.last_copied_content = None
20          self.clipboard_lock = threading.Lock()
21          self.setup_ui()
22          self.start_monitoring()
23          self.start_keyboard_listener()
24
25      def setup_ui(self):
26          self.frame = ttk.Frame(self.root, padding="10")
27          self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
28          self.root.columnconfigure(0, weight=1)
29          self.root.rowconfigure(0, weight=1)
30
31          self.tree = ttk.Treeview(self.frame, columns=("Indicador", "Contenido")
                ↪ , show="headings")
32          self.tree.heading("Indicador", text="")
33          self.tree.heading("Contenido", text="Contenido del Portapapeles")
34          self.tree.column("Indicador", width=30, stretch=tk.NO)
35          self.tree.column("Contenido", width=400, stretch=tk.YES)
36          self.tree.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
37          self.frame.columnconfigure(0, weight=1)
38          self.frame.rowconfigure(0, weight=1)
39
40          scrollbar = ttk.Scrollbar(self.frame, orient=tk.VERTICAL, command=self.
                ↪ tree.yview)
41          scrollbar.grid(row=0, column=1, sticky=(tk.N, tk.S))
42          self.tree.configure(yscrollcommand=scrollbar.set)
43
44          self.tree.bind("<Motion>", self.on_mouse_over)
45          self.tree.bind("<Leave>", self.on_mouse_leave)
46
47          button_frame = ttk.Frame(self.frame)
48          button_frame.grid(row=1, column=0, pady=10)
49
50          font_button = ttk.Button(button_frame, text="Aumentar tamaño de fuente"
                ↪ , command=self.increase_font_size)
51          font_button.grid(row=0, column=0, padx=5)
52
53          clear_button = ttk.Button(button_frame, text="Borrar historial",
                ↪ command=self.clear_history)
54          clear_button.grid(row=0, column=1, padx=5)
55
56          self.status_var = tk.StringVar()
57          status_label = ttk.Label(self.frame, textvariable=self.status_var)
58          status_label.grid(row=2, column=0, pady=5)
59
60          self.update_font()
61          self.setup_styles()
62
63      def setup_styles(self):
64          style = ttk.Style()
65          style.configure("Treeview", rowheight=30)
66          style.configure("Treeview", background="white", fieldbackground="white"
                ↪ , foreground="black")
67          style.map("Treeview", background=[('selected', '#a6a6a6')])
68
69          self.tree.tag_configure('odd', background='#F0F0F0')
70          self.tree.tag_configure('even', background='#FFFFFF')
71          self.tree.tag_configure('gray_circle', foreground='gray')
72          self.tree.tag_configure('green_circle', foreground='lime')
73
74      def start_monitoring(self):
```

```
        self.monitoring_thread = threading.Thread(target=self.monitor_clipboard
            ↪ , daemon=True)
        self.monitoring_thread.start()

    def monitor_clipboard(self):
        last_content = ""
        while True:
            try:
                with self.clipboard_lock:
                    content = self.get_clipboard_content()
                if content != last_content:
                    self.root.after(0, self.add_to_history, content)
                    last_content = content
            except Exception as e:
                print(f"Error al monitorear el portapapeles: {e}")
            time.sleep(0.5)

    def get_clipboard_content(self):
        formats = [
            (win32con.CF_UNICODETEXT, self.get_unicode_text),
            (win32con.CF_TEXT, self.get_text),
            (win32con.CF_BITMAP, self.get_image_text),
            (win32con.CF_HDROP, self.get_file_text)
        ]

        try:
            win32clipboard.OpenClipboard()
            for format, getter in formats:
                if win32clipboard.IsClipboardFormatAvailable(format):
                    content = getter()
                    if content:
                        return content
            return "Contenido no soportado"
        except Exception as e:
            print(f"Error al obtener contenido del portapapeles: {e}")
            return "Error al leer el portapapeles"
        finally:
            win32clipboard.CloseClipboard()

    def get_unicode_text(self):
        return win32clipboard.GetClipboardData(win32con.CF_UNICODETEXT)

    def get_text(self):
        return win32clipboard.GetClipboardData(win32con.CF_TEXT).decode('utf-8'
            ↪ )

    def get_image_text(self):
        return "[Imagen en el portapapeles]"

    def get_file_text(self):
        files = win32clipboard.GetClipboardData(win32con.CF_HDROP)
        return "[Archivos en el portapapeles]: " + ", ".join(files)

    def set_clipboard_content(self, content):
        try:
            win32clipboard.OpenClipboard()
            win32clipboard.EmptyClipboard()
            win32clipboard.SetClipboardText(content, win32con.CF_UNICODETEXT)
        except Exception as e:
            print(f"Error al establecer contenido en el portapapeles: {e}")
        finally:
            win32clipboard.CloseClipboard()

    def start_keyboard_listener(self):
        def on_press(key):
            if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
                self.ctrl_pressed = True
            elif hasattr(key, 'char') and key.char == 'c' and self.ctrl_pressed
```

```
                ↪ :
141                 self.root.after(100, self.update_on_ctrl_c)
142
143         def on_release(key):
144             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
145                 self.ctrl_pressed = False
146
147         self.ctrl_pressed = False
148         listener = keyboard.Listener(on_press=on_press, on_release=on_release)
149         listener.start()
150
151     def update_on_ctrl_c(self):
152         with self.clipboard_lock:
153             content = self.get_clipboard_content()
154             self.add_to_history(content)
155
156     def add_to_history(self, content):
157         if content and content not in self.clipboard_history:
158             self.clipboard_history.appendleft(content)
159             self.update_treeview()
160
161     def update_treeview(self):
162         self.tree.delete(*self.tree.get_children())
163         for i, content in enumerate(self.clipboard_history):
164             item = self.tree.insert("", "end", values=("*", content[:50] + "...
                ↪ " if len(content) > 50 else content),
165                                     tags=('odd' if i % 2 else 'even', '
                                        ↪ gray_circle'))
166
167     def on_mouse_over(self, event):
168         item = self.tree.identify_row(event.y)
169         if item != self.hover_item:
170             self.hover_item = item
171             self.hover_time = time.time()
172         elif item and time.time() - self.hover_time >= 1:
173             content = self.tree.item(item, "values")[1]
174             if content.endswith("..."):
175                 content = list(self.clipboard_history)[self.tree.index(item)]
176
177             if content != self.last_copied_content:
178                 with self.clipboard_lock:
179                     self.set_clipboard_content(content)
180                 self.root.update()
181                 pyautogui.hotkey('ctrl', 'v')
182                 self.last_copied_content = content
183                 self.status_var.set(f"Copiado: {content[:30]}...")
184                 self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + (
                    ↪ 'green_circle',))
185                 self.root.after(2000, lambda: self.reset_status_and_color(item)
                    ↪ )
186             else:
187                 self.status_var.set("Contenido ya copiado.")
188                 self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + (
                    ↪ 'green_circle',))
189                 self.root.after(2000, lambda: self.reset_status_and_color(item)
                    ↪ )
190
191     def on_mouse_leave(self, event):
192         self.hover_item = None
193         self.hover_time = 0
194         self.status_var.set("")
195
196     def reset_status_and_color(self, item):
197         self.status_var.set("")
198         self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
            ↪ gray_circle',))
199
200     def increase_font_size(self):
```

```
        self.font_size += 2
        self.update_font()

    def update_font(self):
        style = ttk.Style()
        style.configure("Treeview", font=("TkDefaultFont", self.font_size))
        style.configure("Treeview.Heading", font=("TkDefaultFont", self.
            ↪ font_size, "bold"))

    def clear_history(self):
        self.clipboard_history.clear()
        self.update_treeview()
        self.status_var.set("Historial borrado")
        self.root.after(100, self.clear_clipboard)

    def clear_clipboard(self):
        with self.clipboard_lock:
            try:
                win32clipboard.OpenClipboard()
                win32clipboard.EmptyClipboard()
                win32clipboard.CloseClipboard()
            except Exception as e:
                print(f"Error al limpiar el portapapeles: {e}")

if __name__ == "__main__":
    root = tk.Tk()
    app = ClipboardHistoryViewer(root)
    root.mainloop()
```

Bibliografía

[1] Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

[2] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: A quantitative approach*. Elsevier.

[3] Rauber, T., & Rünger, G. (2013). *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media.

[4] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485).

[5] Gorelick, M., & Ozsvald, I. (2014). *High performance Python: Practical performant programming for humans*. O'Reilly Media, Inc.

[6] Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

[7] Liu, X., & Chen, T. (2011). Survey of real-time processing systems for big data. *Procedia Engineering*, 15, 3750-3754.

[8] Goetz, B., & Peierls, T. (2006). *Java concurrency in practice*. Pearson Education.

[9] Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, 31(5), 532-533.

[10] Li, D., de Supinski, B. R., Schulz, M., Nikolopoulos, D. S., & Cameron, K. W. (2013). Thread-level energy efficiency optimization for multi-threaded applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 178-187). IEEE.

[11] Herlihy, M., & Shavit, N. (2011). *The art of multiprocessor programming*. Morgan Kaufmann.

[12] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 202-210.

[13] Tanenbaum, A. S., & Van Steen, M. (2016). *Distributed systems: Principles and paradigms*. Prentice-Hall.

[14] Drepper, U. (2007). What every programmer should know about memory. *Red Hat, Inc.*, 11.

[15] Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems*. Pearson.

[16] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*. Wiley.

[17] Stallings, W. (2018). *Operating systems: Internals and design principles*. Pearson.

[18] Love, R. (2013). *Linux kernel development*. Addison-Wesley Professional.

[19] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books.

[20] Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel: From I/O ports to process management*. O'Reilly Media, Inc.

[21] Beazley, D. M. (2010). *Python essential reference*. Addison-Wesley Professional.

[22] Ramalho, L. (2015). *Fluent Python: Clear, concise, and effective programming*. O'Reilly Media, Inc.

[23] Python Software Foundation. (2022). multiprocessing — Process-based parallelism. `https://docs.python.org/3/library/multiprocessing.html`

[24] Python Software Foundation. (2022). asyncio — Asynchronous I/O. `https://docs.python.org/3/library/asyncio.html`

[25] Smith, S., & Wouters, E. V. (2022). PEP 703 – Making the Global Interpreter Lock Optional in CPython. `https://peps.python.org/pep-0703/`

[26] Hellmann, D. (2011). *The Python standard library by example*. Addison-Wesley Professional.

[27] Python Software Foundation. (2023). threading — Thread-based parallelism. `https://docs.python.org/3/library/threading.html`

[28] Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2), 115-138.

[29] Dijkstra, E. W. (1968). Cooperating sequential processes. In *The origin of concurrent programming* (pp. 65-138). Springer.

Direcciones de correo de los autores

Salas-García, Javier: jsalasg@uaemex.mx

Portillo-Rodríguez, Otniel: oportillor@uaemex.mx

Valle-Cruz, David: dvallec@uaemex.mx

Moran-Gonzalez, Miguel Armando: miguel@poilower.com