Facultad de Informática de BarcelonA, UPC

Optimización de Redes con Búsqueda Local

Autores:
Eulàlia Peiret, Marc Teixidó i Eudald Pizarro

Profesor: Carles Fenollosa

Q2 2024-25

Índice

Abstract

En este trabajo, abordamos el problema de optimizar la transmisión de datos desde sensores hasta centros de datos mediante técnicas de búsqueda local. Para ello, exploramos y comparamos los algoritmos *Hill Climbing* y *Simulated Annealing*, evaluando su desempeño en la construcción de una red eficiente. Nuestra investigación incluyó experimentación con diferentes configuraciones y parámetros para determinar parámetros ofrecen mejores resultados en la transmisión de datos.

1 Descripción del problema

En la práctica, se nos pide resolver un problema de interconexión entre una red de sensores y un conjunto de centros de datos. El objetivo principal es implementar un sistema capaz de almacenar toda la información capturada por los sensores mientras se optimiza al máximo el coste asociado. Este trabajo lo hemos tratado como un problema de búsqueda local.

Esencialmente el algoritmo de búsqueda local consiste en:

1. **Selección de una solución inicial**: Se elige una solución x del conjunto de soluciones factibles S, es decir, $x \in S$. Esta solución se considera la solución actual.

2. **Aplicación de operadores**: A partir de la solución actual x, se generan soluciones succesoras y mediante la aplicación de operadores específicos, donde $y \in S$.

3. **Evaluación y actualización**: Dependiendo del criterio del algoritmo, se selecciona una nueva solución y, ya sea porque su función heurística *f(y)* mejora *f(x)* o por ser la primera encontrada. Cuando la nueva solución es aceptada, se convierte en la nueva solución actual.

4. **Iteración**: Los pasos 2 y 3 se repiten hasta que se alcanza un criterio de parada, como la ejecución de un número máximo de iteraciones o que la solucion cumpla algun criterio concreto.

El problema se plantea con búsqueda local ya que el conjunto de soluciones S puede llegar a ser extremadamente grande si aumentamos la cantidad de sensores y centros de datos. Una exploración exhaustiva del espacio de soluciones es inviable por tiempo y recursos computacionales. La búsqueda local permite explorar solo una porción del espacio, encontrando soluciones de calidad en un tiempo más razonable.

Aunque la búsqueda local no garantiza encontrar el óptimo global, es efectiva para encontrar óptimos locales de alta calidad. En problemas como el nuestro, un óptimo local puede ser una solución aceptable y práctica.

A continuación vamos a analizar en detalle los elementos del problema:

1.1 Entrada del problema

Tenemos una área geográfica de 100×100 km^2 con S sensores distribuidos y C centros de datos.

Cada sensor captura datos a velocidades de 1, 2 o 5 Mb/s y puede almacenar hasta el doble del volumen de datos que genera por segundo. Para transmitir su información, cada sensor puede establecer una conexión y enviar todos los datos almacenados durante un segundo, con un máximo de tres conexiones activas simultáneamente.

Los centros de datos reciben la información transmitida por los sensores, con una capacidad máxima de recepción de 150 Mb/s. Además, cada centro de datos puede gestionar hasta 25 conexiones activas al mismo tiempo.

Cada sensor y cada centro están en unas coordenadas del área que se plantea y las conexiones tienen un coste asociado que se calcula a partir de la siguiente fórmula:

$$\text{coste}(x, y) = d(x, y)^2 \times v(x)$$

donde:

- $d(x, y)$ es la distancia euclidiana entre dos puntos.
- $v(x)$ es el volumen de datos transmitidas.

Se nos proporcionan implementadas las clases **Sensores** y **CentrosDatos**. Estas clases pueden instanciarse indicando en sus parámetros la cantidad de sensores o centros de datos que deseamos generar, junto con una semilla para la generación aleatoria. Por ejemplo, al realizar la llamada:

```
Sensores s = new Sensores(10, 1);
```

se crea una instancia de la clase **Sensores**, que es una extensión de `java.util.ArrayList<Sensor>`. Dado que **Sensores** hereda de **ArrayList**, podemos utilizar los métodos proporcionados por esta clase para acceder y manipular los elementos de la lista.

Cada objeto de tipo **Sensor** cuenta con una serie de métodos que permiten obtener sus características, tales como:

- *getCapacidad()*: Devuelve la capacidad del sensor.
- *getCoordX()*: Devuelve la coordenada x del sensor.
- *getCoordY()*: Devuelve la coordenada y del sensor.

De manera similar funciona la clase **CentrosDatos**. Cada centro de datos también dispone de métodos para obtener sus coordenadas.

1.2 Salida del problema

Queremos resolver el problema para una instancia concreta. En la solución, debemos determinar con quién debe conectarse cada sensor y obtener una red lo más eficiente posible, respetando siempre las siguientes restricciones:

- Todos los sensores deben estar conectados a los centros de datos, ya sea directamente o a través de otros sensores.
- Si un sensor no puede almacenar todos los datos recibidos, los datos excedentes se pierden, pero el costo de la transmisión se computa igualmente.
- No se pueden superar los límites de conexiones (3 por sensor y 25 por centro de datos).
- Se debe minimizar el costo total de la transmisión.
- Se debe maximizar la cantidad de datos transmitidos.

2 Implementación del problema

Hemos representado el problema como un grafo dirigido acíclico (DAG), donde cada nodo representa un sensor o un centro de datos y cada arista representa una conexión entre nodos. Para ello, hemos desarrollado la clase **Nodo**, de la cual heredan **NSensor** y **NCentro**.

La clase **NSensor** representa los nodos correspondientes a los sensores y cuenta con los siguientes atributos:

- *sensor*: Instancia del sensor correspondiente.
- *idDestino*: Identificador del nodo al que transmite la información. coste: Coste asociado a la transmisión de datos.
- *infoEnviable*: Cantidad de información (en Mb/s) que el sensor puede enviar. Este atributo es útil para determinar si un nodo está enviando el máximo de información posible o si aún tiene capacidad disponible.
- *recibidos*: Lista (ArrayList) con los identificadores de los sensores de los que recibe datos.

La clase **NCentro** representa los nodos correspondientes a los centros de datos y tiene los siguientes atributos:

- *centro*: Instancia del centro de datos representado.
- *recibidos*: Lista (ArrayList) con los identificadores de los nodos de los que recibe información.
- *informacionRecibida*: Cantidad total de datos (en Mb/s) que está recibiendo el centro de datos en la configuración actual.

En nuestro modelo, un estado está representado por un array de nodos (elementos de la clase **Nodo**). Cada nodo almacena la información necesaria para determinar quién transmite a quién y la configuración de la red en un instante dado.

Para almacenar la información de las conexiones, utilizamos listas de adyacencia dinámicas (ArrayList) en cada nodo. Esta elección es básicamente por dos motivos:

- **Eficiencia en memoria**: Dado que cada sensor puede tener como máximo 3 conexiones y cada centro de datos hasta 25, una representación con una matriz de adyacencias ocuparía mucho más espacio del necesario.
- **Flexibilidad**: El tamaño de las listas puede cambiar dinámicamente en cada iteración del algoritmo, lo que facilita la actualización y modificación de la red.

2.1 Estado inicial

Para solucionar el problema, partimos de un estado inicial, que en este caso consiste en una solución completa. Esto plantea desafíos importantes como encontrar dicha solución con un costo computacional razonable o hacer que el punto inicio sea bueno para la búsqueda de mejoras. La función encargada de generar este estado inicial recibe como parámetros una instancia de **Sensores**, una de **CentrosDatos** y un tercer argumento que define el **método** a utilizar para construir la solución inicial. Se han definido tres enfoques distintos para realizar esta asignación: Greedy (Voraz), Proximidad y Random.

El método **Greedy** asigna los sensores a los centros de datos priorizando siempre la menor distancia. Para ello, recorre todos los sensores y, para cada uno, selecciona el centro más cercano. Esta asignación tiene un costo de $O(c \cdot s)$, donde **s** representa el número de sensores y **c** el número de centros de datos. Una vez realizada esta asignación inicial, se ejecuta un segundo recorrido para verificar que cada centro no tenga más de 25 sensores asignados. En caso de que un centro supere este límite, se busca otro disponible. Si no existe ninguno con capacidad, el sensor se conecta al sensor más cercano que no genere ciclos. Este proceso de verificación también tiene un costo de $O(c + s \cdot (c + 2s))$.

Hemos elegido el método Greedy porque nuestro principal objetivo es garantizar que los datos lleguen a los centros de datos de la manera más directa posible. Al asignar cada sensor al centro más cercano, evitamos la formación de cadenas largas de sensores retransmitiendo información, lo que en un escenario real, podría generar retrasos y congestión en la red ya que una posible avería en un sensor afectaría significativamente el flujo de datos.

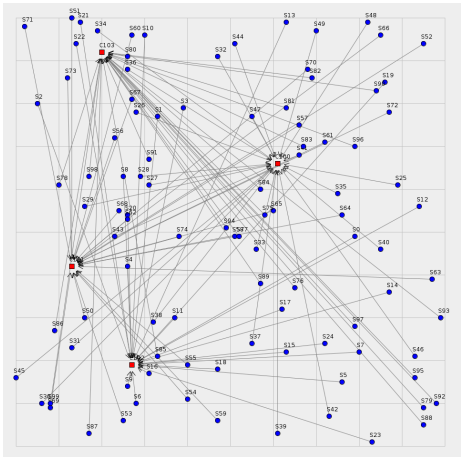

Figure 1: Solución inicial Greedy para 4 centros y 100 sensores

El segundo método, basado en la **proximidad**, genera la solución inicial asignando a cada sensor su nodo más cercano, ya sea un centro de datos u otro sensor. En este caso, se respetan las restricciones de capacidad: un sensor solo puede conectarse a un máximo de tres sensores, mientras que los centros pueden recibir hasta 25 sensores. Para cada sensor, se calcula la distancia con todos los demás nodos y se asigna al más próximo que cumpla con las restricciones. Dado que este proceso considera tanto centros como sensores en la búsqueda del nodo más cercano, el costo computacional de este método es $O(s \cdot (c + s) \cdot (c + 2s))$.

Este método permite distribuir mejor la carga de transmisión y aprovechar al máximo la capacidad de los sensores. Además, minimiza el costo de transmisión al reducir la distancia total recorrida por los datos. Sin embargo, su principal inconveniente es que, en muchos casos, no logra transmitir una cantidad significativa de datos.

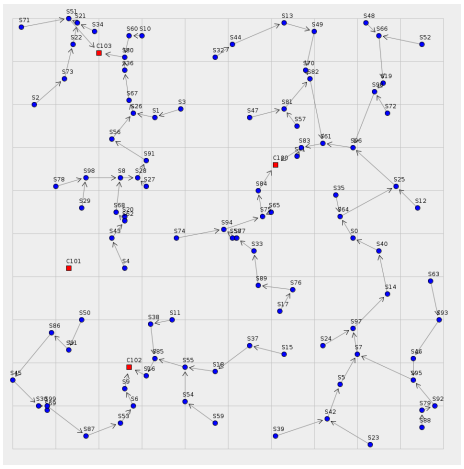

Figure 2: Solución inicial Proximidad para 4 centros y 100 sensores

El tercer enfoque es el método **aleatorio**, donde cada sensor es asignado a un destino elegido al azar, que puede ser tanto un centro de datos como otro sensor. A pesar de su naturaleza aleatoria, se asegura que la asignación cumpla con las restricciones establecidas, evitando ciclos y respetando los límites de capacidad. Para cada sensor, se selecciona un destino al azar y, en caso de que no sea válido, se repite el proceso hasta encontrar una asignación adecuada. Debido a que la validación de restricciones involucra tanto sensores como centros de datos, el costo computacional de este método es $O(s \cdot (c+s) \cdot (c+2s))$. El método **Random** se ha desarrollado con fines experimentales y de comparación. Al generar soluciones iniciales aleatorias, podemos analizar cómo los algoritmos de optimización responden en diferentes escenarios y evaluar su capacidad para mejorar soluciones desde distintos puntos de partida.

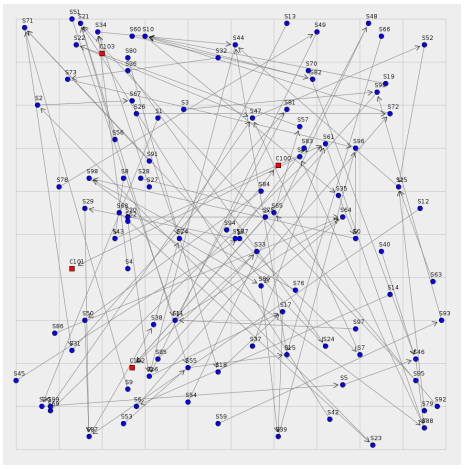

Figure 3: Solución inicial Random para 4 centros y 100 sensores

2.2 Estado final

En este caso, lo que nos interesa es la calidad del estado final. Para ello, necesitamos una función que nos permita ordenar los estados según un criterio determinado. En nuestro caso, este criterio es el valor que devuelve heurístico y nuestro objetivo es minimizarlo al máximo.

La finalización de la búsqueda dependerá completamente del algoritmo utilizado. No existe un estado final predefinido, sino que será el propio algoritmo el que determine cuándo no es posible encontrar una mejor solución.

En el caso de *Hill Climbing*, el algoritmo aceptará la mejor solución encontrada en cada iteración y continuará explorando hasta que no haya ningún estado descendiente con una calidad superior a la solución actual.

Por otro lado, *Simulated Annealing* ejecutará un número determinado de iteraciones según los parámetros establecidos.

2.3 Espacio de búsqueda

El espacio de soluciones del problema está compuesto por todas las posibles formas en que los S sensores pueden enviar los datos a los C centros de datos disponibles. Cada solución válida representa una distribución específica de los datos desde los sensores hasta los centros de datos, respetando las restricciones de conectividad.

Cada sensor solo puede enviar datos a un único destino y los centros no envian datos, lo que limita el número de conexiones posibles en la red. Como máximo, habrá S conexiones en la red, ya que hay S sensores.

Dado que estamos trabajando con un grafo dirigido acíclico (DAG), las conexiones entre los sensores y los centros de datos deben cumplir con las propiedades de un DAG, es decir, no debe haber ciclos. En el peor de los casos, si todas las soluciones respetaran las condiciones de conectividad y no se violaran las restricciones, el número total de soluciones válidas podría ser tan grande como el número de grafos DAG posibles que se pueden formar con S + C nodos (los S sensores y los C centros de datos) y S aristas (una por cada sensor). Esta es una cota superior del espacio de soluciones.

2.4 Operadores que permiten modificar los estados

Los operadores de cambio de la solución permiten modificar una solución completa sin un costo computacional significativo. Estas operaciones permiten la exploración del espacio de soluciones al reconfigurar las conexiones entre sensores y centros de datos.

Se han definido tres operadores fundamentales:

- **swap(int id1, int id2)** Este operador intercambia las conexiones de salida de dos sensores identificados por *id1* y *id2*. Antes de la operación, el sensor *id1* puede estar enviando información a un nodo *id3*, mientras que el sensor *id2* se encuentra conectado a un nodo *c1* (que puede ser otro sensor o un centro de datos). Tras ejecutar *swap(int id1, int id2),* el sensor *id1* se conectará a *c1*, y el sensor *id2* se conectará a *id3*.
  - Restricciones:
    - Se debe verificar que el intercambio no genere ciclos en la red.

    - Los dos ids (*id1* y *id2*) han de pertenecer a sensores ya que los centros no transmiten datos.

  - Factor de ramificación: El factor de ramificación de este operador es el número de posibles combinaciones de sensores que se pueden intercambiar. En términos combinatorios, el número de combinaciones posibles es

$$\binom{S}{2}$$

    donde S es el número de sensores. Esto se debe a que estamos eligiendo 2 sensores entre los S disponibles para realizar el intercambio. Si tenemos S sensores, el factor de ramificación será

$$\frac{S(S-1)}{2}$$

    en el caso más general, ya que cada par de sensores podría intercambiar sus conexiones.

- **connect(int id1, int id2)** Establece una conexión directa entre el sensor *id1* y el nodo *id2*. Si *id1* ya estaba enviando información a otro nodo, esa conexión previa se elimina y se reemplaza por la nueva conexión hacia *id2*.

  - Restricciones:

    - Se debe verificar que el intercambio no genere ciclos en la red.

    - *id2* debe tener capacidad para aceptar una nueva conexión (es decir, no debe estar ya al máximo de conexiones permitidas).

  - Factor de ramificación: El factor de ramificación de este operador depende de los posibles destinos a los que *id1* puede conectarse. Si todos los nodos del grafo pudieran aceptar una conexión, el factor de ramificación es de $(s \cdot (c + s - 1))$.

- **alibera(int id1, int id2)** Este operador tiene como objetivo redirigir todas las conexiones entrantes del nodo *id1* hacia el nodo *id2*. Después de su ejecución, todos los sensores o nodos que enviaban información a *id1* ahora estarán conectados a *id2*, dejando *id1* sin conexiones entrantes. Este operador cambia muchas conexiones a la vez, esto puede ser útil para que algoritmos como Hill Climbing puedan salir de óptimos locales.

  - Restricciones:

    - Se debe verificar que el intercambio no genere ciclos en la red.

    - *id2* debe tener capacidad para aceptar todas las conexiones que llegan a *id1*.

  - Factor de ramificación: El factor de ramificación de este operador depende de los posibles destinos a los que *id1* puede conectarse. Si todos los nodos del grafo pudieran aceptar una conexión, el factor de ramificación es de $((c + s) \cdot (c + s - 1))$.

2.5 Análisis de la función heurística

En esta práctica se emplean los algoritmos de *Hill Climbing* y *Simulated Annealing*. Estos algoritmos utilizan una función heurística para guiar su búsqueda y determinar el siguiente movimiento.

La función heurística es una medida de calidad que orienta la exploración del espacio de soluciones, dado que en los problemas de búsqueda local no existe un estado final definido.

Para nuestra implementación, hemos diseñado la función heurística considerando dos factores clave:

- La cantidad de datos transmitidos.
- El coste de la transmisión.

Ambos criterios tienen la misma importancia, ya que, en un escenario real, los usuarios querrían recibir la información de manera rápida y económica, evitando costes excesivos.

Por ello, la función heurística se define como:

$$heurística = infoPerdida \times costeConexiones$$

Somos conscientes que el valor que tomará *infoPerdida* es mucho más pequeño que *costeConexiones*.Pero en los casos que estudiamos (típicamente con 4 centros y 100 sensores), los centros tienen capacidad suficiente para almacenar toda la información generada por los sensores. Esto significa que, en la mayoría de estados explorados la información perdida (infoPerdida) tiende a ser muy baja (cercana a cero), ya que casi todos los datos pueden llegar a algún centro. Y al hacer esta multiplicación sin factores de conversión, se evita que el algoritmo se conforme con soluciones subóptimas (aunque todos los datos lleguen, sigue buscando configuraciones más baratas). Mantiene la presión para optimizar costes, incluso cuando la información ya está garantizada.

3 Experimentación

Para llevar a cabo todos los experimentos propuestos, hemos dividido el trabajo entre los integrantes, ejecutando el proyecto en diferentes ordenadores. Esto puede haber afectado levemente los resultados, especialmente al comparar los tiempos de ejecución entre distintos experimentos.

Sin embargo, hemos intentado mantener un entorno lo más parecido posible para analizar con precisión cómo influyen los diferentes parámetros en la calidad de las soluciones obtenidas.

Cuando hablamos del *coste* o *valor* de una solución, nos referimos al número devuelto por la función heurística, el cual representa una combinación del coste de transmisión de los datos y el coste asociado a la información perdida. Explicación más detallada de esto en el apartado 2.5.

3.1 Experimentación con conjunto de operadores

La elección de un buen conjunto de operadores es fundamental para resolver problemas de búsqueda local. En este experimento, nos enfocamos en analizar cuál es la mejor combinación de operadores para el algoritmo *Hill Climbing*.

El objetivo es ver qué operadores permiten llegar a una mejor solución, es decir, minimizar la información perdida y el coste de transmisión de la red. Sin embargo, somos conscientes de que *Hill Climbing* es un algoritmo que, en cada iteración, expande todos los posibles sucesores y selecciona

el mejor. Con lo que si el conjunto de operadores es demasiado grande, el tiempo de cómputo puede aumentar exponencialmente, aunque la calidad de la solución mejore.

Para este estudio, veremos también si el impacto de los operadores cambia en funcion de la solucion inicial de la que partimos.

Observación	Puede haber conjuntos de operadores con los que obtienes mejores soluciones.
Planteamiento	Experimentamos con todas las combinaciones de operadores y observamos los resultados.
Hipótesis	Como más operadores usemos, mejor solución obtendremos, pero más tardará en encontrarla.

Table 1: Experimento de operadores

Método:

- Fijaremos el número de centros a 4 y el número de sensores a 100.
- Usaremos el algoritmo *Hill Climbing*.
- La función heurística será la explicada en el apartado 2.5.
- Elegiremos 10 semillas aleatorias y ejecutaremos 1 experimento por cada semilla.
- Repetiremos las 10 pruebas para cada conjunto de operadores y para cada solución inicial.

La etiqueta valor en las gráficas representa el valor de la heuristica en la solución final.

Resultados partiendo de solución inicial de Proximidad:

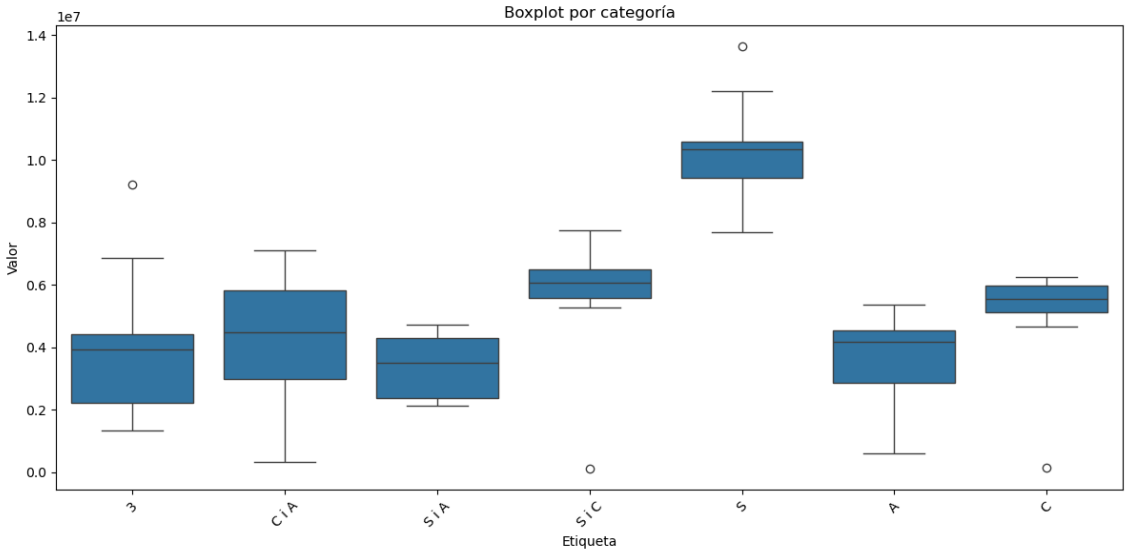

Figure 4: Boxplot de calidad respecto al conjunto de operadores usados para proximidad.

Los conjuntos de operadores más buenos serían:

- La etiqueta 3, que representa el uso de los tres operadores simultáneamente.
- El uso de swap y conecta.
- El uso de swap y liberar.
- El uso de liberar.

Resultados partiendo de solución inicial Greedy:

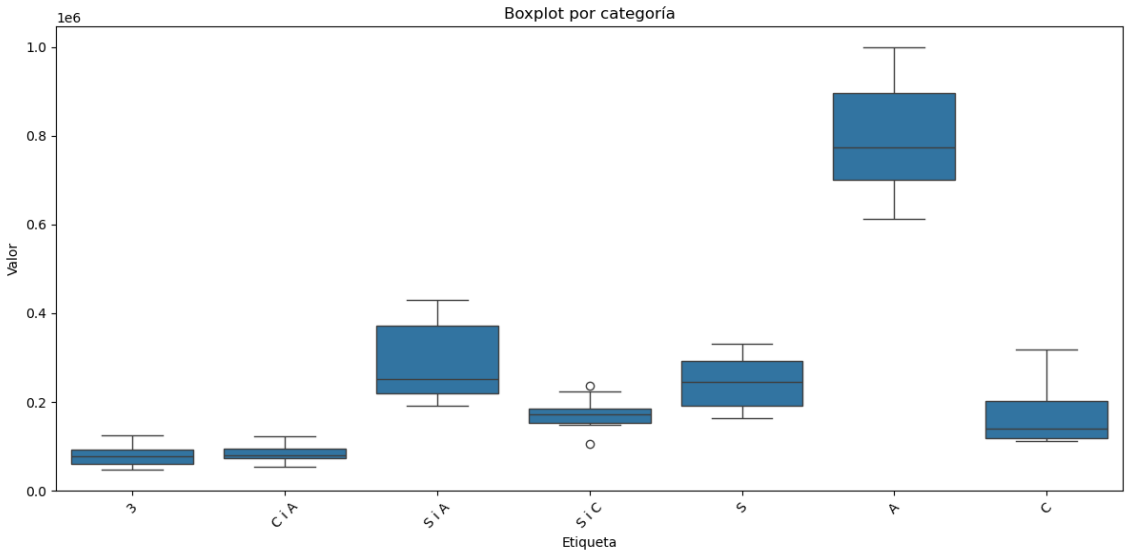

Figure 5: Boxplot de calidad respecto al conjunto de operadores usados para greedy.

Los conjuntos de operadores más buenos serían:

- La etiqueta 3, que representa el uso de los tres operadores simultáneamente.
- El uso de conecta y liberar.
- El uso de swap y conecta.
- El uso de swap y liberar.
- El uso de conecta.

Resultados partiendo de solución inicial Random:

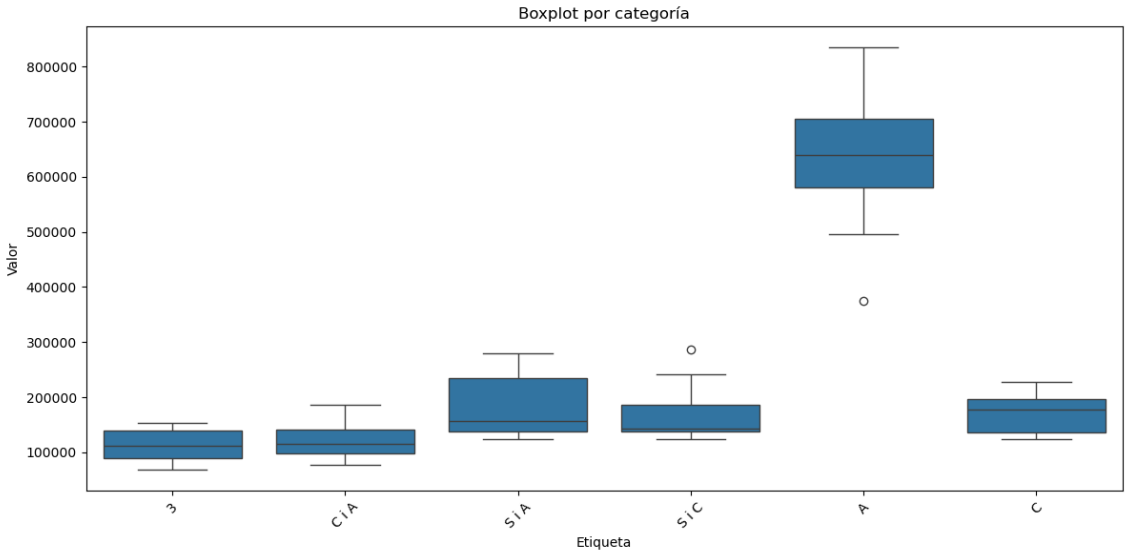

Figure 6: Boxplot de calidad respecto al conjunto de operadores usados para random.

Los conjuntos de operadores más buenos serían:

- La etiqueta 3, que representa el uso de los tres operadores simultáneamente.
- El uso de conecta y liberar.
- El uso de swap y conecta.
- El uso de swap y liberar.
- El uso de conecta.

Conclusión: Para determinar qué conjunto de operadores funciona mejor también hemos tenido en cuenta el tiempo que tardan (en milisegundos) en generar la solución:

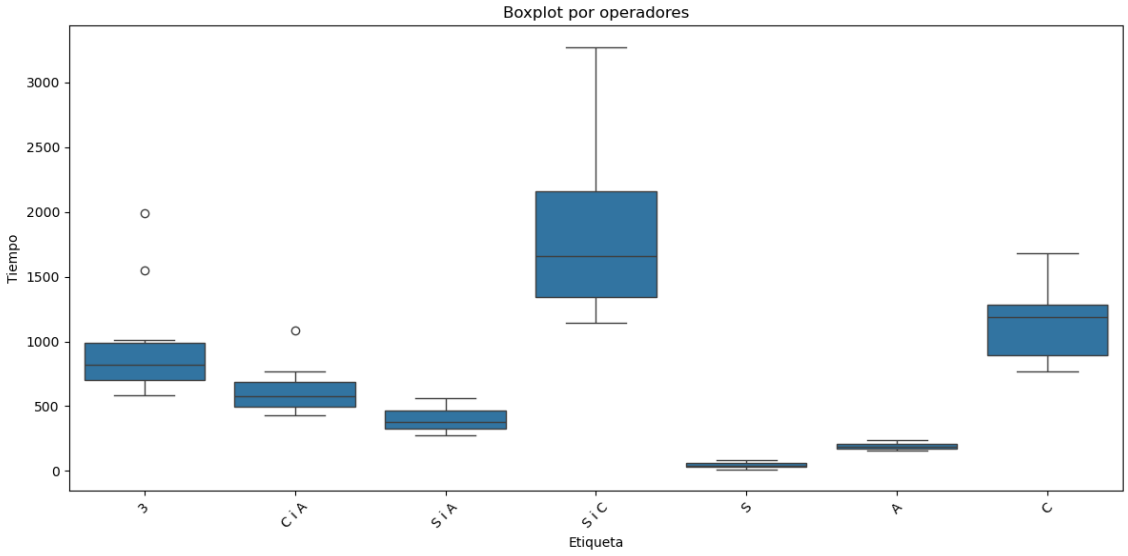

Figure 7: Boxplot de tiempo en ms respecto al conjunto de operadores usados para proximidad.

Independientemente del estado inicial, al medir los tiempos de ejecución, los resultados son bastante similares.

Como esperábamos en nuestra hipótesis, la combinación de los tres operadores juntos siempre genera estados de buena calidad. Sin embargo, usar los tres operadores simultáneamente es de las opciones que más tiempo tarda en dar un resultado. Esto tiene sentido, ya que, como mencionamos previamente, el algoritmo de *hill climbing* expande todos los nodos posibles en cada iteración, y cuantos más operadores se utilicen, más nodos podrán generarse.

Por lo tanto aún sabiendo que con los dos operadores escogidos no recorremos todo el espacio de soluciones, teniendo en cuenta tanto la calidad de las soluciones como los tiempos de ejecución, consideramos que la combinación más adecuada de operadores es **Swap** y **Liberar**. Ya que independientemente de no explorar todo el espacio de soluciones tiene mucha probabilidad de darnos una muy buena solución en mucho menos tiempo. A parte esto és bueno ya que como veremos en el experimento 4 el tiempo según el tamaño de entrada cambiará drasticamente así que es positivo empezar con el menor tiempo posible.

En consecuencia, para los siguientes experimentos, utilizaremos este conjunto de operadores.

3.2 Experimentación con las soluciones iniciales

La elección de una buena estrategia para generar un estado inicial es fundamental para poder explotar bien todo el potencial de los algoritmos de búsqueda local. En este experimento, nos centramos en analizar cómo afecta la elección del estado inicial a la calidad de la solución final.

El objetivo es determinar cuál de las tres estrategias de generación de estados iniciales (Greedy, Proximidad y Aleatorio) es la más adecuada para el algoritmo de *Hill Climbing*. Para este experimento se usará la misma heurística que en el experimento anterior y partiremos de que solo tenemos los operadores de swap y liberar, ya que se ha demostrado que son los que mejor funcionan. A partir de los resultados de este experimento, se fijará la estrategia de generación de los estados iniciales para los siguientes experimentos.

Observación	La calidad de la solución final depende significativamente del estado inicial.
Planteamiento	Se compararán las tres estrategias de generación de estados iniciales (Greedy, Proximidad y Random) utilizando el algoritmo Hill Climbing.
Hipótesis	La estrategia de proximidad nos permitirá obtener mejor estado final.

Table 2: Experimento con los estados iniciales

Método

- Fijaremos el número de centros a 4 y el número de sensores a 100.
- Usaremos el algoritmo *Hill Climbing*.
- La función heurística será *costeTransmisión * infoPerdida*.
- Elegiremos 10 semillas aleatorias y para cada semilla haremos:
  - Una ejecución con la estrategia Greedy.
  - Una ejecución con la estrategia Proximidad.
  - Cinco ejecuciones con la estrategia Random y nos quedaremos con la media.
- Usaremos los operadores de swap y liberar.
- Mediremos la calidad de la solución final en función de la heurística.

Se espera que la inicialización por proximidad sea la mejor estrategia, ya que asigna los sensores a los centros de datos o sensores más cercanos, lo que reduce el coste de transmisión. Se espera que la estrategia Greedy obtenga soluciones con menos información perdida, pero con un coste de transmisión más elevado. Por último, la estrategia Random debería obtener soluciones intermedias, ya que asigna los sensores de forma aleatoria, lo que puede llevar a soluciones con más información perdida y un coste total más elevado.

Resultados y conclusiones

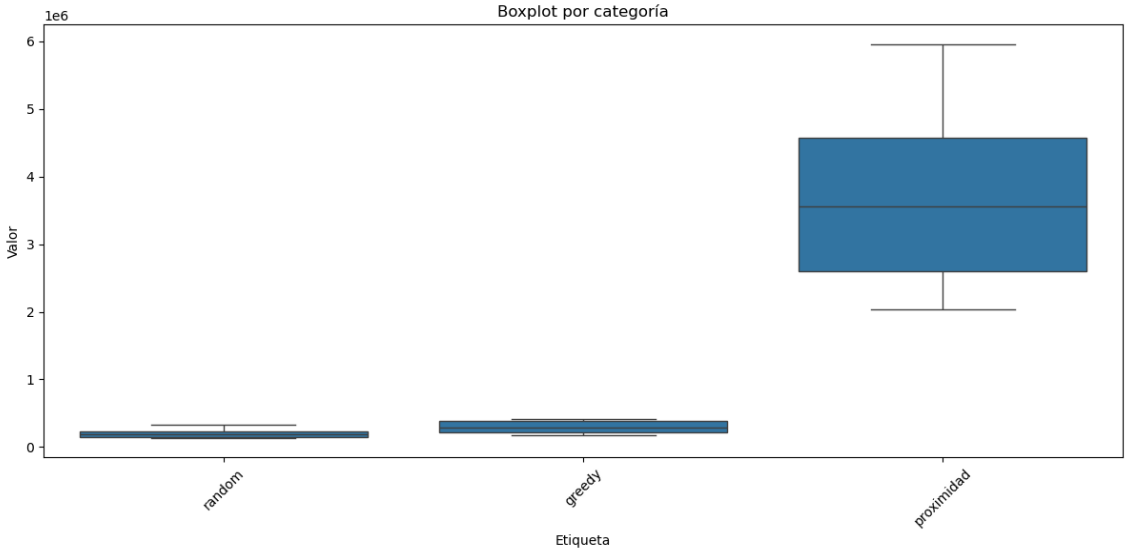

Figure 8: Comparación de calidad(valor) con diferentes soluciones iniciales

Los resultados nos han sorprendido mucho ya que la mejor estrategia es la Random. Creemos que esto puede ser ya que al ser aleatoria permite explorar más el espacio de soluciones, es decir, nos permite explotar más los operadores, por tanto, llegamos a mejores soluciones. Pero se tiene que decir que esta estrategia es la más costosa en términos de tiempo de ejecución.

3.3 Experimentación con los parámetros del Simulated Annealing

El algoritmo Simulated Annealing se diferencia del Hill Climbing en su enfoque para la exploración del espacio de soluciones. Mientras que Hill Climbing siempre selecciona el mejor vecino en cada paso, lo que puede llevarlo a quedarse atrapado en óptimos locales, Simulated Annealing usa una estrategia basada en probabilidades para permitir la exploración de soluciones menos óptimas en ciertos momentos.

En lugar de expandir todos los posibles estados vecinos, Simulated Annealing selecciona aleatoriamente un descendiente en cada iteración. Luego, compara la nueva solución generada con la solución actual:

1. Si la nueva solución es mejor, se acepta directamente.
2. Si la nueva solución es peor, se puede aceptar dependiendo de una probabilidad calculada en base a la temperatura.

La temperatura es un parámetro clave del algoritmo y va disminuyendo gradualmente a lo largo de la ejecución. Al principio, cuando la temperatura es alta, hay una mayor probabilidad de aceptar soluciones peores, lo que permite explorar el espacio de búsqueda de manera más amplia. A medida

que la temperatura baja, el algoritmo se vuelve más restrictivo y se enfoca en la explotación de soluciones cercanas al óptimo.

Los parámetros con los que hemos experimentado son:

- *steps* (Número total de iteraciones): Define cuántas veces se generará una nueva solución antes de que el algoritmo se detenga.
- *stiter* : Número de iteraciones realizadas a una temperatura específica antes de reducirla.
- k : Constante utilizada en la ecuación de probabilidad para aceptar soluciones peores.
- *lambda* (λ): Factor que controla la tasa de enfriamiento de la temperatura.

3.3.1 Experimentación con el número de iteraciones

Observación	La calidad de las soluciones obtenidas cambia en función del número de iteraciones.(El coste de las graficas corresponde a la calidad).
Planteamiento	Experimentamos con el número de iteraciones y observamos los resultados.
Hipótesis	El número de iteraciones será directamente proporcional a la calidad de la solución.

Table 3: Experimento con el parámetro *steps*

Método: Para determinar el número óptimo de iteraciones en el algoritmo Simulated Annealing, fijamos los demás parámetros en valores estándar:

- $\lambda = 0.01$
- $k = 5$
- $stiter = 100$

La variable *steps* empieza siendo 10 y la incrementamos hasta 5000, con incrementos de 50 por prueba. De cada prueba hacemos la media de 10 experimentos con semillas distintas. Las pruebas se han hecho con 100 sensores y 4 centros.

Resultados:

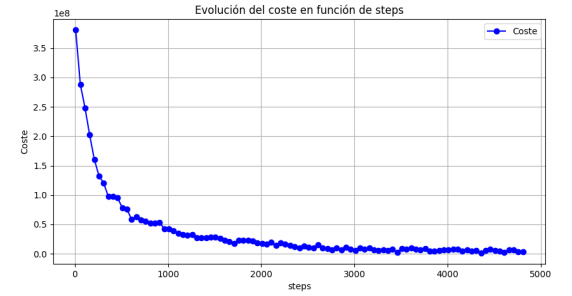

Figure 9: Coste en función de la variable *steps*

Los resultados obtenidos nos muestran que, con un número reducido de iteraciones, el algoritmo encuentra soluciones de mucho coste. A medida que el número de iteraciones aumenta, el coste de la solución mejora, estabilizándose cuando se alcanzan aproximadamente 3000 iteraciones.

Esto indica que, con estos parámetros, después de 3000 iteraciones, la temperatura ha descendido lo suficiente como para que el algoritmo haya explorado una parte significativa del espacio de búsqueda. A partir de este punto, aumentar el número de iteraciones no genera mejoras significativas en la calidad de la solución, porque ya es muy óptima.

Conclusiones: Nuestra hipótesis inicial era que tendrían una relación directamente proporcional la calidad de la solución y la variable *steps*. No ha sido exactamente así. El impacto del número de iteraciones en la calidad de las soluciones es especialmente notable en el rango de 0 a 1000 iteraciones, donde se observa una mejora considerable de la calidad de las soluciones. Entre 1000 y 3000 iteraciones, la calidad sigue aumentando, aunque con un margen de mejora cada vez menor. Finalmente, a partir de 3000 iteraciones, el algoritmo encuentra una solución altamente óptima, lo que sugiere que continuar iterando más allá de este punto ofrece un beneficio mínimo.

Teniendo en estos resultados, establecemos el número óptimo de iteraciones en 3000, asegurando un buen balance entre calidad de solución y tiempo de ejecución.

3.3.2 Experimentación con la variable *stiter*

Observación	La calidad de las soluciones obtenidas cambia en función a los parámetros.
Planteamiento	Le damos diferentes valores a la variable *stiter* y observamos los resultados.
Hipótesis	El con stiter muy grandes, la calidad de la solución será peor porque la temperatura no podrá bajar mucho.

Table 4: Experimento con el parámetro *stiter*

Método: Para determinar el valor óptimo de *stiter*, fijamos los siguientes parámetros con base en los resultados obtenidos en la sección anterior:

- $\lambda = 0.01$
- k = 5
- *steps* = 3000

El parámetro *stiter* lo inicializamos en 10 y se lo incrementamos de 50 en 50 hasta alcanzar un valor de 5000. De cada valor de stiter haremos diez pruebas con semillas distintas y nos quedaremos con la media de ellas. Siempre en una red de 100 sensores y 4 centros.

Resultados y conclusiones:

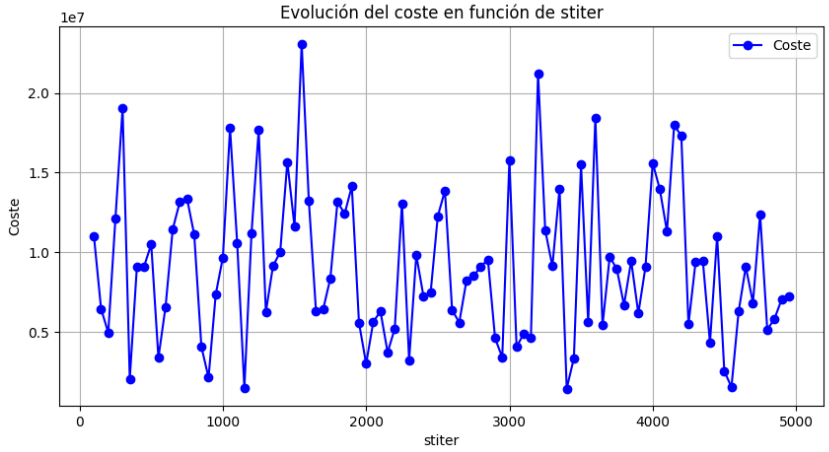

Figure 10: Calidad de las soluciones en función de la variable *stiter*

El análisis de los resultados no muestra una relación clara entre *stiter* y la calidad de la solución como esperábamos en la hipótesis. Establecemos el valor de stiter = 500 como el más adecuado para nuestras pruebas, ya que parece ser un punto que minimiza el coste aunque en los resultados no vemos indicios de que el algoritmo tenga en cuenta esta variable.

3.3.3 Experimentación con la variable λ

Observación	La calidad de las soluciones obtenidas cambia en función a los parámetros.
Planteamiento	Experimentamos con λ. Después comentamos los resultados.
Hipótesis	El parámetro λ hará que la temperatura disminuya más rápido o lento. Si la temperatura disminuye muy rápido, se necesitaran muchas más iteraciones para obtener un buen resultado.

Table 5: Experimento con el parámetro λ

Método:

Para determinar el valor óptimo de λ, fijamos los siguientes parámetros basándonos en los resultados obtenidos hasta ahora:

- $stiter = 500$
- $steps = 3000$
- $k = 5$

Experimentaremos con valores de λ desde 0.01 hasta 1 incrementando 0.05 en cada experimento. Como siempre, por cada valor de λ realizamos 10 pruebas con semillas distintas y nos quedamos con la media de ellas. Todo ello en un escenario de 100 sensores y 4 centros.

Resultados y conclusiones:

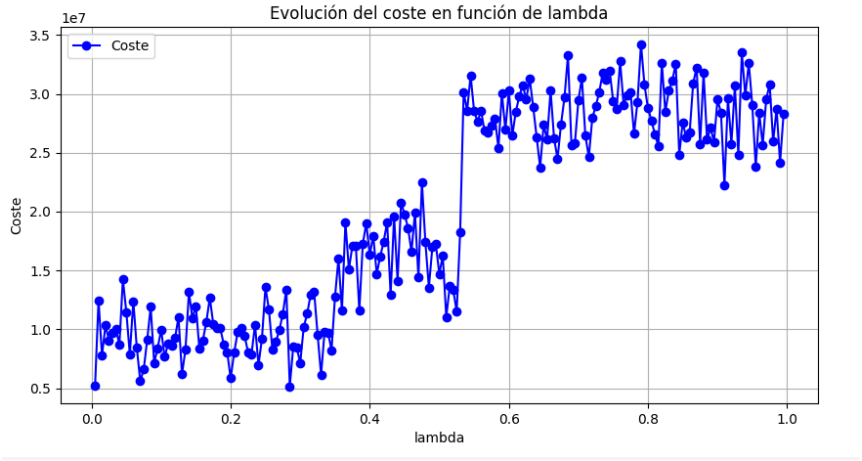

Figure 11: Coste en función del parámetro λ

La hipótesis parece confirmarse, ya que con valores de λ más pequeños se obtienen mejores soluciones, con un coste menor. En particular, se observa que cuando λ supera el valor de 0.5, la calidad de la solución empeora significativamente. También pensábamos que, aunque el número de λ fuera alto, al aumentar el número de *steps* podríamos llegar a soluciones igualmente buenas. Sin embargo, al experimentar con diferentes valores de λ y *steps*, se ha comprobado que, independientemente del número de *steps*, siempre es más eficiente trabajar con valores pequeños de λ . Además, el cambio en la calidad de la solución cuando λ supera 0.5 se mantiene constante para distintos valores de *steps*. Por lo tanto, se ha decidido fijar el parámetro *lambda* en 0.01.

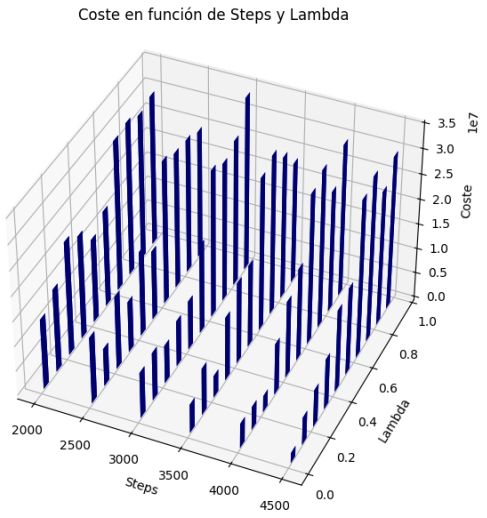

Figure 12: Coste en función de λ y *steps*

3.3.4 Experimentación con la variable *k*

Observación	La calidad de las soluciones obtenidas cambia en función a los parámetros.
Planteamiento	Experimentamos con k. Después comentamos los resultados.
Hipótesis	El parámetro k afectará a los resultados.

Table 6: Experimentación con el parámetro k

Método:

Con los experimentos realizados anteriormente, hemos determinado que los parámetros que ofrecen mejores resultados son los siguientes:

- stiter = 500
- Iteraciones = 3000
- $\lambda = 0.01$

A continuación, probaremos distintos valores de k en el rango de [5, 200], incrementando su valor en 5 unidades por cada experimento. Para cada valor de k, realizaremos 10 pruebas y calcularemos la media de los resultados obtenidos.

Resultados y conclusiones:

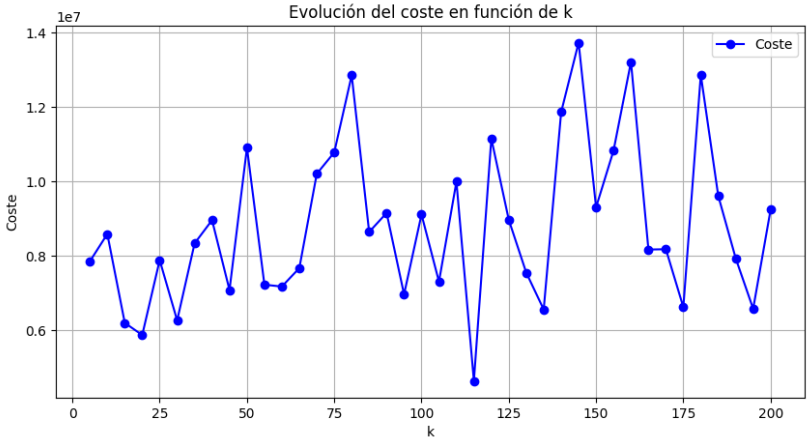

Figure 13: Coste en función del parámetro k

Aunque no se observa una tendencia clara al variar el parámetro k, se puede destacar que con valores más pequeños no se obtienen soluciones excesivamente malas.

En conclusión, para este experimento con 100 sensores y 4 centros, utilizando los operadores swap y liberar, el algoritmo de Simulated Annealing y la generación de estados iniciales random, los parámetros que proporcionan la mejor solución son los siguientes:

- $stiter = 500$
- $steps = 3000$
- $\lambda = 0.01$
- $k = 25$

3.4 Experimentación con el tiempo de ejecución

Observación	Cuando se augmenta el número de centros y de sensores hace que el espacio de soluciones aumente.
Planteamiento	Experimentamos cambiando el número de centros y sensores. Después comentamos los resultados.
Hipótesis	El augmento en tiempo respecto a la cantidad de centros y sensores será exponencial.

Table 7: Experimentación con el número de centros y sensores

Método:

- Empezaremos el número de sensores a 100.
- Usaremos el algoritmo *Hill Climbing*.
- La solución inicial será generada con el método random. (los establecidos en el experimento 3.2)
- Los operadores usados son el de Swap y el de Libera. (los establecidos en el experimento 3.1)
- Experimentaremos con los siguientes números de centros [4,6,8,10,12,14,16,18] con los respectivos números de sensores [100,150,200,250,300,350,400,450] respetando la proporción 4:100.
- Para cada par de centros-sensores se ejecutan 10 pruebas con 10 semillas aleatorias y se hace el promedio del tiempo de ejecución.

Resultados: Haciendo esto conseguimos aproximar a una función de apariencia exponencial vista en esta imagen:

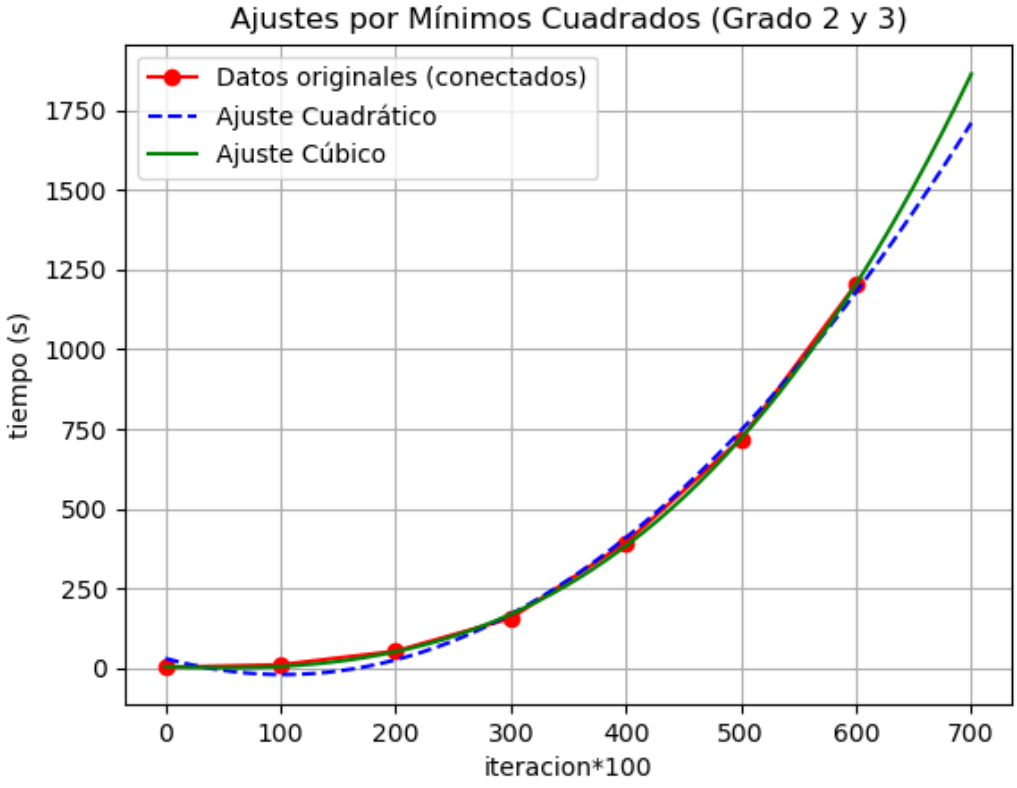

Figure 14: Función aproximada del tiempo para cada iteración en segundos

Para ver la tendencia de la función, la hemos aproximado usando el método de ajustes por mínimos cuadrados y hemos encontrado estas dos posibles funciones:

- $0.004849x^2 - 0.9928x + 30.02$ **Cuadrática**
- $4.263e-06x^3 + 0.001012x^2 - 0.1402x + 4.441$ **Cúbica**

Conclusiones: Se confirma nuestra hipótesis: el tiempo de ejecución parece que crece de manera exponencial a medida que aumenta el tamaño del problema. Esto tiene sentido, ya que *Hill Climbing* genera muchos más estados sucesores, dado que el factor de ramificación de los operadores depende directamente del número de sensores y centros. Por lo tanto, en cada iteración, se podrán generar muchos más sucesores, lo que hace que el espacio de soluciones sea mayor y, en consecuencia, tarde más en llegar al punto en el que ninguno de los sucesores mejora la solución actual.

De todas formas, no podemos asegurar con total certeza que el crecimiento sea estrictamente exponencial, ya que no pudimos realizar más iteraciones debido a limitaciones de memoria. Llegado cierto punto, el programa dejaba de ejecutarse por rendimiento y memoria. Sin embargo, con los datos que hemos recopilado y la aproximación obtenida, observamos una clara tendencia exponencial en el crecimiento del tiempo de ejecución.

3.5 Experimentación con las dimensiones del problema

En los escenarios anteriores, hemos trabajado con un número fijo de centros de datos y sensores, siendo 4 y 100, respectivamente. Sabemos que la capacidad de recepción de cada centro de datos es de 150 Mb/s, lo que permite una capacidad total de recepción de 600 Mb/s. Por otro lado, los sensores tienen una capacidad máxima de envío de 265 Mb/s en total. El objetivo de este experimento es determinar si, en este escenario, los sensores utilizan todos los centros de datos disponibles o no.

Observación	La capacidad de los centros de datos es mucho mayor que la de los sensores.
Planteamiento	Analizar si todos los centros de datos son usados en la solución final.
Hipótesis	Para obtener la solución óptima se deben usar todos los centros de datos disponibles.

Table 8: Experimentación con las dimensiones del problema

Método:

- Fijamos el número de sensores a 100.
- Fijamos el número de centros a 4.
- Usaremos el algoritmo *Hill Climbing*.
- La solución inicial será generada con el método random. (establecido en el experimento 3.2)
- Los operadores usados son el de Swap y el de Libera. (los establecidos en el experimento 3.1)
- Se ejecutarán 10000 pruebas con semillas aleatorias.

Resultados y conclusiones:

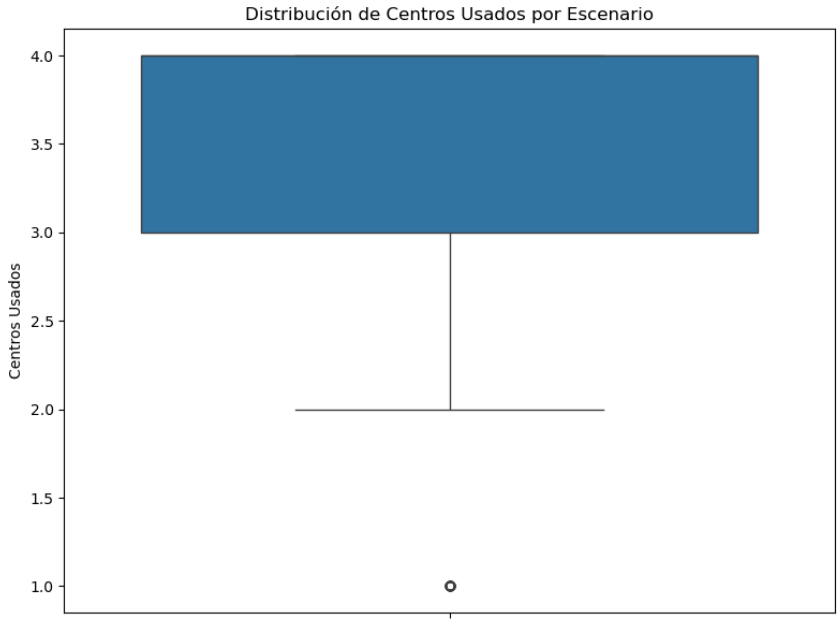

Figure 15: Función aproximada del tiempo para cada iteración en segundos

En la imagen podemos observar que la mayoría de las soluciones finales utilizan todos los centros de datos, es decir, que se usan los 4. Sin embargo, los resultados pueden hacer pensar que no es necesario usar todos los centros de datos.

Nuestra hipótesis inicial era que todos los centros de datos deberían ser utilizados para la mejor solución. Pero al observar los resultados, sí que es cierto que más de la mitad de las soluciones los utilizan todos (el 56 % de ellas), pero también hay un número significativo de soluciones que solo utilizan 3 centros (37 %). Lo que nos lleva a la conclusión que no necesariamente se deben usar todos los centros de datos para obtener una buena solución, esto puede deberse a que la capacidad de los centros proporcionalmente es mucho mayor que la de los sensores.

Según nuestros resultados, la media de los centros usados es de 3.5, lo que indica que en la mayoría de casos se usan todos. Podemos estimar una relación óptima entre centro de datos por sensor de entre 1:25 y 1:33. En el siguiente experimento analizaremos si se obtienen mejores resultados al aumentar el número de centros de datos, manteniendo el número de sensores constante.

3.6 Experimentación con el número de centros de datos

Observación	Los centros son un elemento clave para las soluciones ya que el objetivo es poderles hacer llegar todos los datos.
Planteamiento	Experimentamos cambiando el número de centros. Después comentamos los resultados.
Hipótesis	El como más centros haya, mejor será la solucion porque menos problemas de capacidades tendremos.

Table 9: Experimentación con los centros de datos

Método:

- Fijaremos el número de sensores a 100.
- Usaremos el algoritmo *Hill Climbing* y *Simulated Annealing*.
- La solución inicial será generada con el método random. (establecido en el experimento 3.2)
- Los operadores usados son el de Swap y el de Libera. (los establecidos en el experimento 3.1)
- Experimentaremos con los siguientes números de centros [2,4,6,8,10].
- Para cada número de centros ejecutaremos 10 pruebas con *Hill Climbing* y 10 con *Simulated Annealing* (cada una con semilla distinta), nos quedaremos con la media de estas 10 pruebas.
- Los parámetros para *Simulated Annealing* son *steps* = 3000, *stiter* = 500, k = 25, *lambda* = 0.01. (los establecidos en el experimento 3.3).

Resultados y conclusiones:

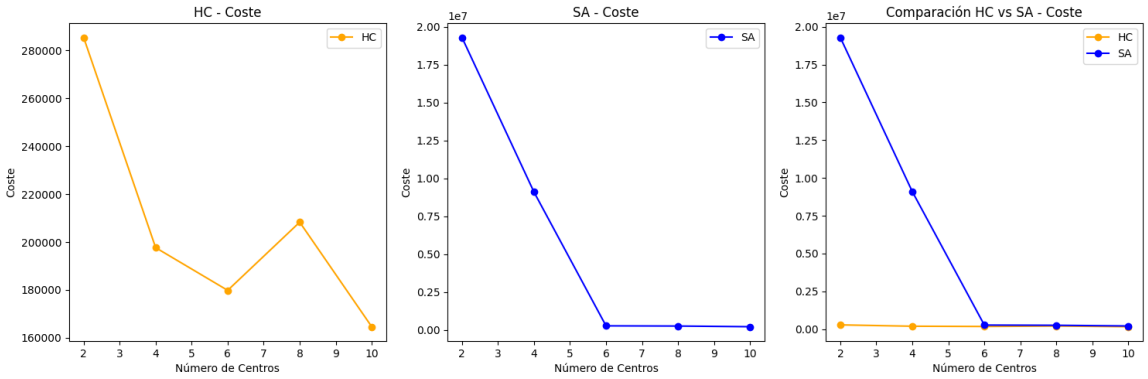

Figure 16: Costes de los algoritmos en función del número de centros de datos

Vemos que nuestra hipótesis se cumple bastante. Aunque a partir de 6 centros, ambos algoritmos encuentran una solución óptima, y el margen de mejora a partir de este punto es muy bajo.

Con un número pequeño de centros, *Simulated Annealing* tiene más dificultades para encontrar una buena solución. En cambio, *Hill Climbing* siempre es capaz de encontrar soluciones con costes muy bajos.

Esto indica que, aunque haya pocos centros, es posible organizar la red de manera eficiente para obtener soluciones de buena calidad. Sin embargo, es probable que haya muchas menos combinaciones que logren un coste bajo, y SA no siempre es capaz de encontrarlas.

A medida que aumenta el número de centros, más combinaciones logran una solución óptima. Como resultado, a partir de 6 centros, ambos algoritmos consiguen encontrar soluciones con costes muy bajos.

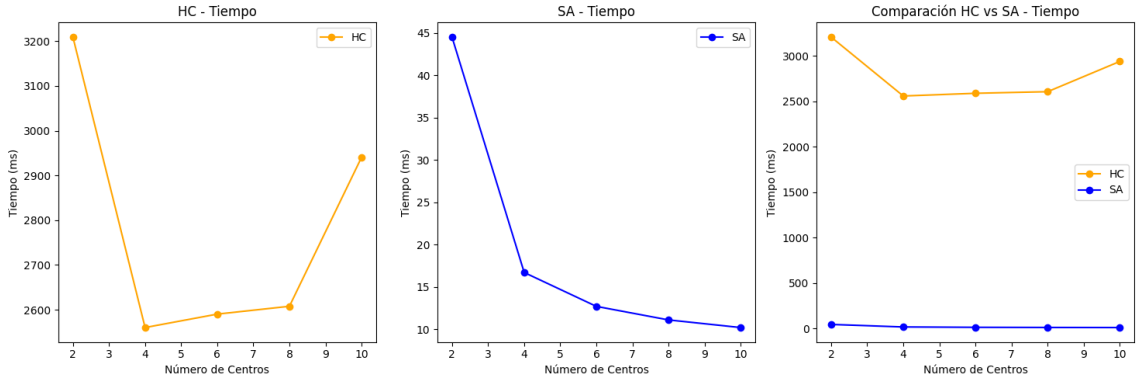

Figure 17: Tiempos con los algoritmos en función del número de centros de datos

En cuanto al tiempo de ejecución, observamos una diferencia considerable entre ambos algoritmos. Independientemente del número de centros, *Simulated Annealing* encuentra una solución mucho más rápido, *Hill Climbing* puede llegar a tardar hasta 64 veces más. Esto se debe al coste computacional de expandir todos los sucesores en cada iteración.

También es interesante notar cómo H.C. aumenta su tiempo de ejecución de forma notable a partir de 6 centros. Creemos que esto se debe a que, al haber más centros, existen muchas más combinaciones posibles que cumplen con las restricciones en cada iteración. Como resultado, el algoritmo debe expandir muchos más estados, lo que aumenta el tiempo de cómputo.

Recopilando todo lo visto en este experimento podemos decir que para 100 sensores no vale la pena poner más de 6 centros. Con 100 sensores y 6 centros ambos algoritmos encuentran soluciones de coste muy bajo. Y el algoritmo más rapido es sin duda *Simulated Annealing*.

3.7 Experimentación con la función heurística

En este experimento se pide jugar con la importancia de la informacion perdida y el coste de transmitirla en las soluciones. Por eso vamos a cambiar la heruística:

$$costeTotal = w \times infoPerdida + costeConexiones$$

Observación	La funcion heuristica es la que calcula la calidad de la solución.
Planteamiento	Experimentamos cambiando la importancia de cada elemento del heurístico. Después comentamos los resultados.
Hipótesis	Como más importancia se le de a la información perdida, menos información se perdera pero el precio a pagar será que el coste de transmitirla será mucho más alto.

Table 10: Experimentación con la función heurística

Método:

- Fijaremos el número de sensores a 100 y el número de centros de datos a 2.
- Usaremos el algoritmo *Hill Climbing*.
- La solución inicial será generada con el método random. (establecido en el experimento 3.2)
- Los operadores usados son el de Swap y el de Libera. (los establecidos en el experimento 3.1)
- La función heuristica será = CosteTotal = $w \times$ infoPerdida + costeTransmisión
- Se experimenta con distintos valores de w en el rango [1, 500, 1000, 1500, ..., 10000].
- Para cada ponderación, se realizan 20 ejecuciones y se calcula la media de los resultados.

Resultados y conclusiones:

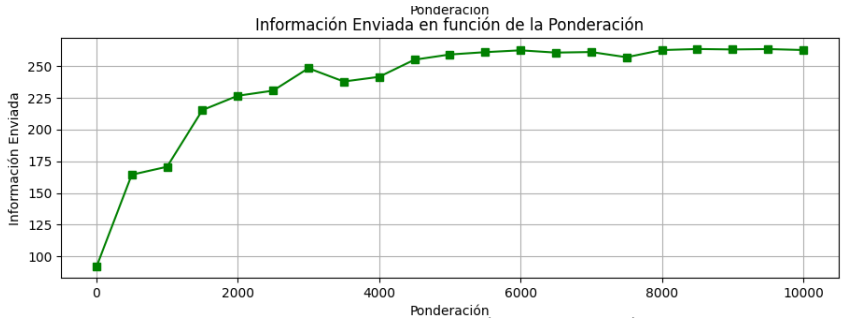

Figure 18: Información enviada en función de la ponderación

Como se planteó en nuestra hipótesis, al aumentar el valor de w, la solución prioriza la minimización de la información perdida, lo que lleva a una mayor cantidad de datos transmitidos. Se observa que a partir de w=400, la cantidad de información enviada apenas mejora, ya que el total de datos emitidos por los sensores es de 266. Al llegar a este punto, prácticamente se está transmitiendo el 100% de la información, por lo que no es posible aumentar más este valor.

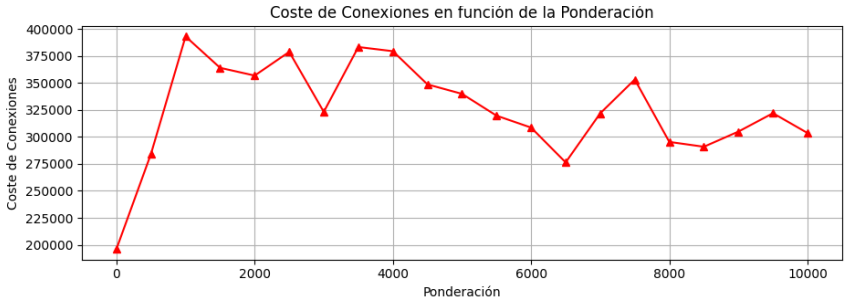

Figure 19: Coste de transmisión en función de la ponderación

En cuanto al coste de transmisión, también se cumple nuestra hipótesis: a medida que se otorga mayor importancia a la información enviada (valores altos de w), el sistema acepta costes de transmisión más elevados. Es decir, la prioridad pasa a ser garantizar la transmisión de toda la información, incluso si esto implica conectar sensores muy distantes entre sí.

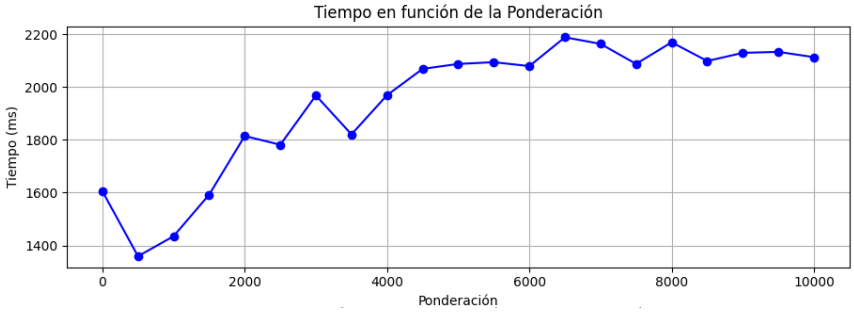

Figure 20: Tiempo de ejecución en función de la ponderación

Respecto al tiempo de ejecución, los valores más altos se registrana partir de w=6000. Esto coincide con los casos en los que se logra un equilibrio entre la cantidad de datos transmitidos y el coste de las conexiones, lo que sugiere que el algoritmo tiene mayor dificultad en optimizar ambos factores simultáneamente.

Por otro lado, cuando w es muy pequeño (es decir, cuando la solución prioriza únicamente la reducción del coste de transmisión), el tiempo de ejecución disminuye significativamente. Esto indica que el algoritmo encuentra soluciones óptimas más rápidamente cuando la única variable de interés es el coste. Auque a pesar de encontrar soluciones rápido, no garantizan ser las más eficientes.

4 Conclusiones

A lo largo de los experimentos, hemos analizado los resultados obtenidos y extraído conclusiones en cada caso. Sin embargo, consideramos relevante hacer un recopilatorio general de todo lo experimentado.

En términos de equilibrio entre tiempo de ejecución y calidad de la solución, la mejor combinación de operadores es Swap-Liberar aunque hemos visto al final que podriamos haber usado Conecta-Liberar que daba ligeramente mejores resultados aunque tardaba aproximadamente entre el doble y el triple de tiempo. Por otro lado, la solución inicial aleatoria (random) permite explorar mejor el espacio de soluciones, aunque su generación es más costosa computacionalmente.

Al comparar los algoritmos, hemos concluido que *Hill Climbing* ofrece mejores soluciones en general. No obstante, este algoritmo es considerablemente más lento y consume mucha más memoria. Este aspecto es crucial, ya que, a medida que aumentan las dimensiones del problema, el tiempo de ejecución crece exponencialmente, lo que puede volverse inviable en escenarios de gran escala.

Como conclusión general, si queremos implementar una red de sensores, es fundamental definir nuestras prioridades y, en función de ellas, seleccionar la estrategia más adecuada.

5 Trabajo de innovación

Nuestro trabajo de innovación se centra en un software de Nexar diseñado para prevenir accidentes de tráfico. Hemos comenzado a investigar sus sistemas y hemos dividido el trabajo en diferentes secciones.

El objetivo de nuestra investigación es profundizar en las siguientes preguntas clave:

¿Qué productos ofrece Nexar?

Nexar ofrece cámaras dashcams que los conductores instalan en sus vehículos, las cuales graban videos de alta definición de todo lo que sucede en la carretera. Estas cámaras están conectadas a la aplicación de Nexar, lo que permite a los usuarios acceder a grabaciones en tiempo real, almacenar automáticamente videos de incidentes y recibir alertas sobre posibles peligros en la vía.

¿Cómo utiliza la inteligencia artificial y qué técnicas emplea?

Actualmente, desde la aplicación de Nexar, las IA son capaces de clasificar videos en los que ocurre algún impacto y guardar automáticamente los videos de incidentes. Nexar está probando y comparando diferentes modelos de inteligencia artificial con el objetivo de mejorar los algoritmos predictivos de colisiones en video. Estos algoritmos analizan secuencias de video grabadas por las dashcams e identifican señales de riesgo en tiempo real. Detectan patrones como frenadas bruscas de vehículos, peatones cruzando sin mirar y distancias peligrosas con otros elementos de la carretera.

¿Qué riesgos y beneficios supone el producto para la empresa y los usuarios?

Entre los beneficios para Nexar, el desarrollo de su sistema de IA y su red de cámaras dashcam le permite recopilar una gran cantidad de datos sobre el tráfico y la conducción, lo que puede mejorar su tecnología y ofrecer servicios innovadores. Es un avance crucial si se quiere dar un paso adelante con la conducción autónoma. Sin embargo, la efectividad de los algoritmos de IA aún tiene limitaciones, y las consecuencias si una IA falla en estas situaciones pueden ser afectar a vidas humanas y tener enormes repercusiones para la empresa y los usuarios.