PRÁCTICA DE BÚSQUEDA LOCAL

INTELIGENCIA ARTIFICIAL

Q1 2024-2025

UNIVERSIDAD POLITÉCNICA DE CATALUÑA

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Natasha Trojan Jimenez

Julia Gallo Escudero

Víctor Moreno Villanueva

Contenido

1. Parte descriptiva

1.1. Descripción del problema

La empresa Ázamon necesita optimizar el envío diario de paquetes a varias ciudades y nos solicita ayuda para hacer más eficiente esta tarea, asegurando que los requisitos establecidos por los clientes y las empresas con las que colaboran quedan satisfechos.

Diariamente, Ázamon recibe ofertas de transporte de diversas empresas de paquetería con diferentes características: capacidad de peso (múltiplo de 5 kg y entre 5 y 50 kg), precio por kilogramo transportado y tiempo de entrega (entre 1 y 5 días).

Los paquetes que deben asignarse tienen las siguientes características: uno de los tres niveles de prioridad que marca la rapidez con la que debe ser entregado (con un coste asociado), y un peso (múltiplo de 0.5 kg y como máximo 10 kg). Tras asignar el paquete a una oferta válida, si debe ser almacenado hasta ser recogido en un día posterior, se incurre un coste de almacenamiento de 0.25 euros por kg y día.

Usando la información de los paquetes y ofertas, Ázamon debe encargarse de asignar todos los paquetes a ofertas respetando su prioridad y optimizando dos factores de rendimiento: los costes y la felicidad. Los costes provienen del coste de almacenamiento y del precio de la entrega (que deben minimizarse), y la felicidad es la suma total de días que se han anticipado los paquetes a su fecha de entrega correspondiente (que debe maximizarse).

Para resolver el problema y realizar las experimentaciones necesarias, hemos definido las siguientes variables globales:

- Conjunto de paquetes que deben asignarse.
- Conjunto de paquetes ordenados según su prioridad.
- Conjunto de ofertas disponibles.
- Registro de la asignación de paquetes a ofertas.
- Registro del peso libre en las ofertas.
- Número de paquetes.
- Número de ofertas.

1.1.1. Elementos del problema

Los elementos más importantes del problema son los paquetes a enviar y las ofertas de las empresas de transporte. Trataremos todos los paquetes como un conjunto único, decidiendo así abstraer las ciudades de nuestra representación para simplificar el estudio. Por lo tanto, los elementos del problema que consideramos son los siguientes:

Ofertas de las empresas de transporte:

- **Peso máximo**: Capacidad de transporte en kg, que oscila entre 5 y 50 kg, en intervalos de 5 kg.
- **Precio por kilogramo**: El coste que cobra cada empresa por transportar cada kilogramo de paquete, variable según la oferta.
- **Tiempo de entrega**: El número de días (entre 1 y 5) que tarda el transporte en llegar al destino.

Paquetes a enviar:

- **Peso de los paquetes**: Los paquetes tienen un peso máximo de 10 kg y son múltiplos de 0,5 kg.
- **Prioridad de entrega**: Los clientes pueden elegir entre tres opciones de prioridad, con diferentes plazos y costes: la prioridad 1 implica la entrega en 1 día, la prioridad 2 en 2 o 3 días, y la prioridad 3 en 4 o 5 días.
- **Coste de almacenamiento**: Los paquetes que tardan más días en ser recogidos generan un coste adicional de 0,25 euros por kilogramo y día.

1.1.2. Búsqueda Local

Para resolver este problema y encontrar una asignación óptima, se utilizará la búsqueda local, que consiste en hallar una solución inicial y explorar sus estados cercanos para encontrar una mejor solución. Esta metodología tiene una complejidad temporal mucho menor en comparación con métodos como el backtracking. La búsqueda local es una herramienta poderosa, siempre que se cumplan las siguientes condiciones:

- Es un problema de optimización pura, cuyo objetivo es obtener el mejor estado según una función que valora su calidad.
- El camino hasta llegar a un estado final no es relevante.

Este problema se beneficia de usar la búsqueda local ya que se cumplen ambas condiciones.

Primero, se busca minimizar los costes (de almacenamiento y de transporte), y maximizar la felicidad. Ambas representan funciones objetivo dependientes del estado que se pueden optimizar. Así, el objetivo es optimizar la asignación de tal manera que se logre el equilibrio entre minimizar costos y maximizar la satisfacción del cliente, lo que es típico de los problemas de optimización pura.

Por otro lado, del resultado solo es relevante la asignación óptima, no las variaciones de la misma que han conducido a ella. El camino a la solución óptima no proporciona información relevante.

Por tanto, al tratarse de un problema de optimización pura, la búsqueda local es una técnica adecuada para encontrar soluciones eficientes sin importar el camino intermedio, sino centrándose en encontrar el mejor estado final posible según los criterios establecidos.

1.1.3. Espacio de búsqueda

El espacio de búsqueda se define como el conjunto completo de posibles soluciones o estados finales que tiene nuestro problema. Es importante conocerlo porque, si el espacio de soluciones tiene un tamaño de orden polinómico, sería factible resolver el problema utilizando fuerza bruta. Además, nos da una idea de la magnitud del problema al que nos estamos enfrentando.

Asumiendo que tenemos n paquetes y m ofertas, consideramos el caso donde cada paquete se puede asignar a cualquiera de las ofertas. Por tanto, para los n paquetes se puede elegir entre m ofertas diferentes, lo que implica un total de m^n combinaciones posibles de asignaciones de paquetes a ofertas.

Pese a que no todas las combinaciones son válidas debido a las restricciones de peso y coste, y por ende el espacio efectivo de búsqueda será menor, la expresión $O(m^n)$ proporciona una buena estimación del potencial de combinaciones a evaluar. Esta estimación es claramente no polinómica, por lo que usar la fuerza bruta no sería productivo.

1.2. Implementación del estado

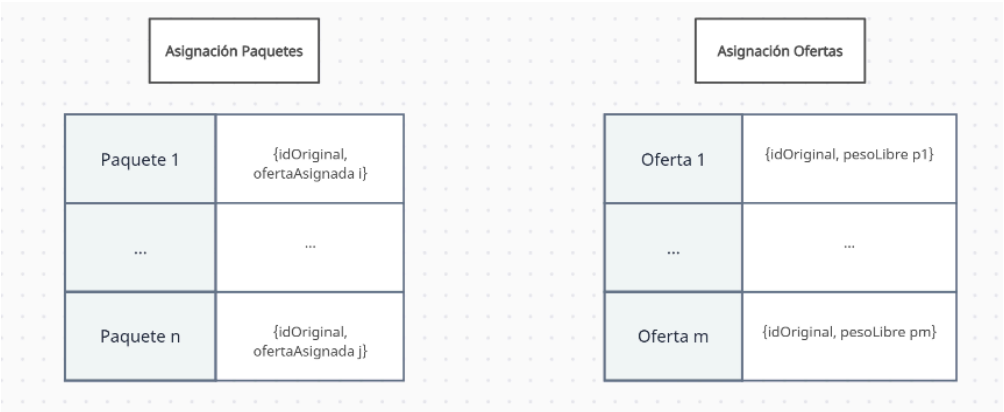
La primera idea intuitiva fue guardar un registro de las asignaciones de paquetes a ofertas, y otro con el peso libre de todas las ofertas. Para n paquetes y m ofertas, la idea consistía en crear dos listas: una lista *asignacionPaquetes* de las n asignaciones de paquetes a ofertas (donde *asignacionPaquetes*[i] devuelve la oferta a la que ha sido asignada el paquete i), y una lista *asignacionOfertas* que guarde el peso libre de cada oferta (donde *asignacionOfertas*[j] devuelve el peso libre en la oferta j). Ninguno de los dos vectores es estático ya que tanto las asignaciones como los pesos libres irán variando con las diferentes soluciones.

El problema que surgió de esta idea fue el de la restricción de prioridades, no se estaba teniendo en cuenta que un paquete solo podía ser asignado a una oferta de su prioridad o de una prioridad mayor.

Entendiendo la "prioridad" de una oferta de la siguiente manera: 0 para las ofertas de transporte para el día siguiente, 1 para las ofertas de 2 o 3 días, y 2 para las ofertas de 4 o 5 días, se barajaron dos posibilidades:

- Comprobar la restricción de prioridad cada vez que se (re)asignaba un paquete.
- Separar las ofertas en tres listas diferentes según su prioridad (*ofertasPrioridad1, ofertasPrioridad2, ofertasPrioridad3*).

Como ambas opciones implican el mismo coste espacial $O(m)$ nos decantamos por la segunda por si fuera necesario utilizar solo uno de los vectores. No obstante, tras varios intentos fallidos y un código excesivamente complicado, decidimos volver a empezar y decantarnos por la primera opción, lo cual simplificó considerablemente el código y dio mejores resultados instantáneamente.



Los paquetes se asignan según su prioridad de mayor prioridad a menor, por lo que al registrar su asignación, además de la oferta a la que se le asigna, se guarda también su id en el conjunto original de paquetes. Por lo tanto, *asignacionPaquetes* es una lista de *Map.Entry<Integer, Integer>*.

Por lo que se refiere a *asignacionOfertas*, se registrará el cambio de su peso libre al añadir/quitar un paquete. Como este cambio no tiene que ocurrir secuencialmente en las ofertas, al guardar el nuevo peso libre se debe registrar junto al identificador de la misma, por lo que *asignacionOfertas* también es una lista de *Map.Entry<Integer, Integer>*.

De esta manera, el coste espacial del estado es $O(2n + 2m) = O(n + m)$, donde *asignacionPaquetes* tendrá coste espacial n y *asignacionOfertas* ocupa un espacio de tamaño m. El hecho de usar listas hizo que las consultas fueran rápidas y eficientes, y facilitó el proceso de añadir y quitar asignaciones debido a su tamaño dinámico.

1.3. Operadores del problema

Para modificar un estado hemos implementado los operadores que se muestran a continuación, y para calcular el factor de ramificación hemos asumido que tendremos *p* paquetes, *o* ofertas y 3 prioridades *pr*.

a) mover paquete con misma prioridad (p1, o2):

- *Parámetros*: el índice del paquete *p1*, la nueva oferta asignada *o2*, y el peso libre resultante en *o2* tras añadir el paquete.
- *Precondiciones*: el peso del paquete *p1* no es mayor al peso libre en la oferta *o2,* el paquete *p1* se encuentra en una oferta distinta a *o2* y se respeta la prioridad del paquete al cambiarlo de oferta.
- *Postcondiciones*: el paquete *p1* pasa a estar asignado a la oferta *o2* y el peso libre de ambas ofertas es actualizado adecuadamente.
- *Factor de ramificación*: Cualquier paquete puede ser asignado a cualquier oferta siempre que la prioridad de la oferta asegure que el paquete llegue en el plazo necesario y el peso restante de la oferta *o2* sea mayor o igual al peso del paquete *p1*, por lo que es del orden de $O(p * o)$.

b) swap paquetes con misma prioridad (p1, p2):

- *Parámetros*: el índice del primer paquete *p1* y el índice del segundo paquete *p2*.

- *Precondiciones*: los dos paquetes tienen la misma prioridad, al intercambiar ambos paquetes no se sobrepasará el peso libre en las ofertas *o2 y o1* respectivamente, los dos paquetes se encuentran en una oferta distinta a la que serán asignados.

- *Postcondiciones*: el paquete *p1* pasa a estar asignado a la oferta *o2,* el paquete *p2* pasa a estar asignado a la oferta *o1* y el peso restante de las dos ofertas *o1 y o2* es actualizado adecuadamente.

- *Factor de ramificación*: Cualquier par de paquetes asignados pueden hacer swap entre ellos siempre que tengan la misma prioridad y el peso libre de las ofertas nuevas sea no se supere al intercambiarlos, por lo que será del orden de $O(p^2)$.

c) mejorar prioridad de un paquete (p1, o2, pr2):

- *Parámetros:* el índice del paquete *p1*, la nueva oferta asignada *o2*, y la prioridad de dicha oferta *pr2.*

- *Precondiciones:* el peso del paquete *p1* no es mayor al peso restante de la oferta *o2,* el paquete *p1* se encuentra en una oferta distinta a *o2* y la prioridad de la oferta *o2* es mejor que la proporcionada por la oferta actual.

- *Postcondiciones:* el paquete *p1* pasa a estar asignado a la oferta *o2* y el peso libre de ambas ofertas es actualizado adecuadamente.

- *Factor de ramificación:* Cualquier paquete puede ser asignado a una oferta *o2* de mayor prioridad siempre que el peso libre de *o2* sea mayor o igual al peso del paquete *p1,* por lo que es del orden de $O(p * o)$.

d) eliminar prioridad (prX) (Eliminada tras experimentación):

- *Parámetros:* el índice prX de la prioridad a eliminar.

- *Precondiciones:* la prioridad a eliminar no puede ser la 0 (mayor prioridad) ya que en el mejor caso tendríamos todos los paquetes en ofertas de esa prioridad, y la prioridad 1 solo se puede eliminar si no hay ningún paquete asignado a una oferta de prioridad 2.

- *Postcondiciones:* la prioridad prX deja de existir y tenemos todos los paquetes asignados a una prioridad mayor, y ninguno en una prioridad peor.

- *Factor de ramificación:* Una prioridad se puede eliminar en tiempo constante pero tiene que cumplir con las condiciones especificadas anteriormente.

Este operador fue descartado debido a su redundancia y poco efecto, ya que no mejora el resultado al usarse. Otro operador que se planteó y se descartó fue el de eliminar ofertas una vez estuvieran vacías, ya que al querer optimizar la felicidad y no el uso de un número mínimo de ofertas no tendría sentido.

En conjunto, los tres operadores (a, b y c) pueden suponer mejoras en el estado, y exploran un espacio de soluciones suficientemente amplio.

1.4. Generación de soluciones iniciales

Para realizar la práctica hemos partido de dos posibles soluciones iniciales:

a) *Solución inicial secuencial*: Para generar esta solución, se itera sobre los paquetes ordenados de mayor a menor prioridad y se hacen las siguientes comprobaciones para las ofertas disponibles en orden secuencial:
   I) La prioridad de la oferta es mayor o igual a la del paquete.
   II) El paquete cabe en la oferta.

   Se asigna el paquete a la primera oferta que cumpla ambos requisitos. Al asignar las ofertas de mayor prioridad primero, se asegura que todos los paquetes estarán asignados al devolver la solución generada.

b) *Solución inicial random:* Se itera sobre la lista de paquetes ordenados según su prioridad (de mayor a menor), y se asigna aleatoriamente a la primera oferta que cumpla con sus requisitos de prioridad y peso, sin seguir el orden de la lista de ofertas.

En resumen, la primera función intenta asignar cada paquete a una oferta recorriendo secuencialmente el vector de ofertas hasta que se puede asignar a una, mientras que la segunda función lleva a cabo la asignación probando ofertas de manera aleatoria en función de restricciones de prioridad y peso.

1.5. Función heurística

Nuestras funciones heurísticas sirven para evaluar la calidad de una solución, y se centran en optimizar el proceso de asignación de paquetes a ofertas. Para ello, utilizan dos criterios para comprobar la calidad de una solución:

1. Minimización de costes:
   1.1. Minimización de los costes de transporte.
   1.2. Minimización de los costes de almacenamiento.
2. Maximización de la felicidad de los clientes.

Nota: Para poder obtener soluciones válidas con las restricciones de este problema, es necesario que la capacidad de transporte de las ofertas sea algo superior al peso total de los paquetes a enviar.

Para calcular la primera, se recorre la solución que se quiere evaluar y se va actualizando el contador de los costes de transporte correspondientes a cada oferta, de tal manera que el valor de la primera función heurística se interpreta como (siendo S la solución evaluada, m el -número total de ofertas, y $p(i)$ el número de paquetes asignados a la oferta i):

$$Costes\ totales\ de\ transporte\ (S) = \sum_{i}^{m} \sum_{j}^{p(i)} peso(j) * precioKg(i)$$

Para la segunda, se recorre la solución que se quiere evaluar y se actualiza el contador de los costes de almacenamiento correspondientes a cada oferta, de tal manera que el valor de la segunda función heurística se interpreta como (siendo S la solución evaluada, m el número total de ofertas, y $p(i)$ el número de paquetes asignados a la oferta i):

$$Costes\ totales\ de\ almacenamiento\ (S) = \sum_{i}^{m} \sum_{j}^{p(i)} 0.25 * diasAlmacen(i) * peso(j)$$

Entendiendo que *diasAlmacen(i)* devuelve los días que un paquete está en el almacén esperando a ser recogido (1 para ofertas de 3 o 4 días, y 2 para ofertas de 5 días).

La función heurística de costes devuelve, por lo tanto, la suma de ambos contadores.

Para calcular la heurística de la felicidad de los clientes, se recorre la solución que se quiere evaluar y se actualiza el contador de los días que se anticipado cada paquete según su prioridad y la oferta a la que se le ha asignado en la solución. De esta manera, la segunda heurística se interpreta como (siendo S la solución evaluada, m el número total de ofertas, y $p(i)$ el número de paquetes asignados a la oferta i):

$$Costes\ totales\ de\ almacenamiento\ (S) = \sum_{i}^{m} \sum_{j}^{p(i)} prioridad(i) - prioridad(j)$$

Nos planteamos hacer una heurística conjunta a través de la división $\frac{heurística\ felicidad}{heurística\ costes}$, ya que queríamos maximizar la felicidad y minimizar los costes, pero fue descartada al ver que no era adecuada ya que habría que ponderar ambas (no tienen las mismas unidades) según su importancia, lo cual implicaría evaluar cuál es más importante priorizar. Además, dificulta la interpretación ya que, por ejemplo, un valor bajo podría implicar felicidad baja, muchos costes o ambas a la vez, y viceversa con un valor alto. Por lo tanto, se descartó esta opción debido a que aceptaría soluciones subóptimas. Nos decantamos por dos heurísticas.

2. Parte experimental

2.1. Primer experimento:

Este experimento pretende comprobar qué conjunto de operadores dan mejores resultados para la función heurística de los costes, para poder fijarlos en el resto de experimentos.

Para ello, se usa el algoritmo de Hill Climbing ya que, al tratar de mejorar la heurística con cada iteración, usa los operadores que conducen al sucesor que obtiene los valores óptimos. Por lo tanto, los operadores que utilice serán los mejores.

Nos hemos decantado por la estrategia de inicialización basada en asignación aleatoria (1. 4. b)) ya que pensamos que será la que más operadores y más variados utilizará dada su variabilidad.

Observación: Usando la misma generación de la solución inicial, los operadores utilizados son similares en todas las ejecuciones.

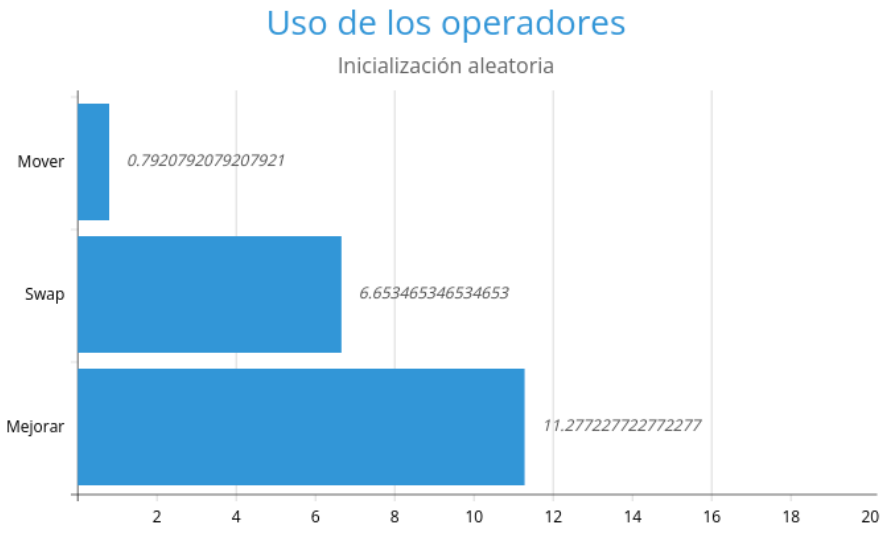
Planteamiento: Cuantificación de los operadores usados por Hill Climbing.

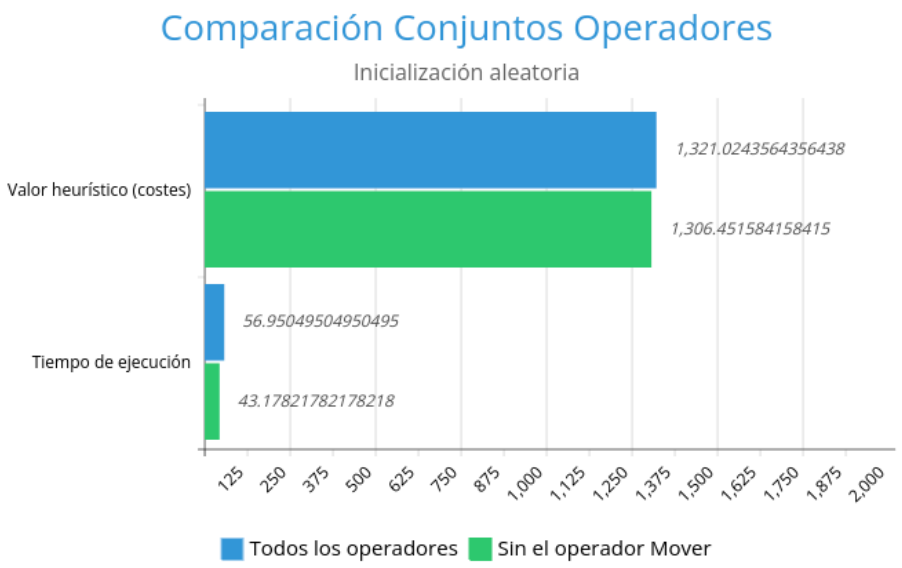
Hipótesis: Al asignar los paquetes de manera aleatoria, el operador más utilizado será el de mejorar prioridad. Aunque el estado inicial sea solución, creemos que la asignación aleatoria asignará paquetes sin llenar las ofertas de una prioridad mayor que quizá mejoran los resultados.

Metodología:

- Parámetros: 100 paquetes y proporción 1,2 de peso transportable por las ofertas.
- Se usa el algoritmo de Hill Climbing y la asignación inicial aleatoria.
- Ejecutamos el programa 100 veces con semillas de generación aleatorias (hasta 2000).
- Calculamos la media de uso de cada operador en todas las ejecuciones para ponderar los resultados en caso de una gran disparidad.

Resultados:





Conclusiones:

Podemos observar que nuestra hipótesis era correcta, ya que el algoritmo ha utilizado el operador Mejorar con mayor frecuencia. También nos damos cuenta de que, pese a que el operador **Mover** casi no es elegido por Hill Climbing, su uso empeora el resultado del experimento (tanto en tiempo como en resultado). Por lo tanto, **queda descartado** para el resto de los experimentos.

2.2. Segundo experimento:

Este experimento pretende comprobar qué generación de la solución inicial da mejores resultados para la función heurística de los costes, usando los parámetros del primer experimento y Hill Climbing, para fijar la generación en los siguientes experimentos.

Observación: No todas las generaciones de la solución inicial dan los mismos resultados.

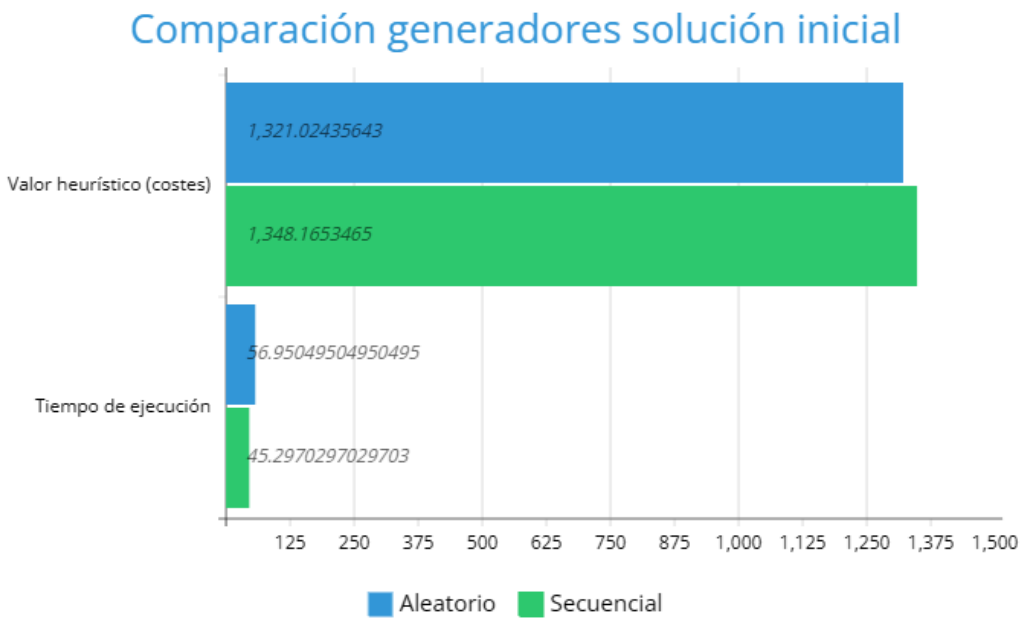
Planteamiento: Comparación de las generaciones de la solución inicial.

Hipótesis: La asignación aleatoria dará mejores resultados porque será más diversa en las ejecuciones, escapa de soluciones locales subóptimas y es más flexible que la secuencial.

Metodología:

- Parámetros: 100 paquetes y proporción 1,2 de peso transportable por las ofertas.
- Se usa el algoritmo de Hill Climbing.
- Ejecutamos el programa 100 veces con semillas de generación aleatorias (hasta 2000).
- Comparamos el valor de la heurística según cada generador de la solución inicial.
- Comparamos también el tiempo de ejecución por si los resultados fueran muy similares.

Resultados:



Conclusiones:

Podemos observar que nuestra hipótesis era correcta, ya que el generador aleatorio ha causado un valor mejor del heurístico. También nos damos cuenta de que la diferencia entre los dos generadores es mínima, tanto por el valor del heurístico como por el tiempo. Por lo tanto, nos decantamos por la **asignación aleatoria** y su mejor resultado para los siguientes experimentos (a pesar de que tarda un poco más).

2.3. Tercer experimento:

Este experimento pretende comprobar qué valor de los parámetros (k, λ y $stiter$) dan mejores resultados para la función heurística de los costes, usando el algoritmo Simulated Annealing.

Observación: Diferentes valores de los parámetros darán resultados diferentes.

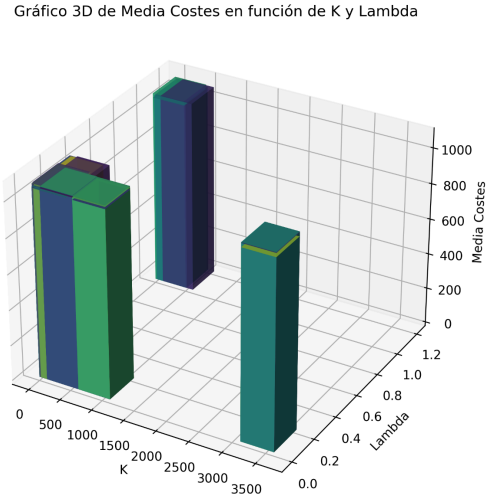
Planteamiento: Prueba de distintos parámetros con tal de encontrar la mejor combinación.

Hipótesis: Nula.

Metodología:

- Parámetros: 100 paquetes y proporción 1,2 de peso transportable por las ofertas.
- Se usa el algoritmo de Simulated Annealing con 10000 pasos.
- Ejecutamos el programa 100 veces con generación de parámetros aleatorios para cada combinación de los siguientes valores:
  - k = [1, 5, 25, 125,625,3125]
  - λ = [1.0, 0.1, 0.01, 0.001, 0.0001]
  - $stiter$ = [10,20,25,50,75,100]
- Calculamos la media de los valores de las 100 ejecuciones de cada combinación, buscando la media máxima.

Resultados:



stiter = 100

Conclusiones:

Los mejores valores son $stiter$ = 100, λ = 0.01, y k = 25, lo cual es razonable teniendo en cuenta que Simulated Annealing no necesita valores muy extremos.

2.4. Cuarto experimento:

Este experimento pretende estudiar cómo evoluciona el tiempo de ejecución según la magnitud del problema (número de paquetes y proporción del peso transportable). Se usa Hill Climbing y la heurística de los costes.

Observación: Una mayor magnitud del problema ralentizará la ejecución.

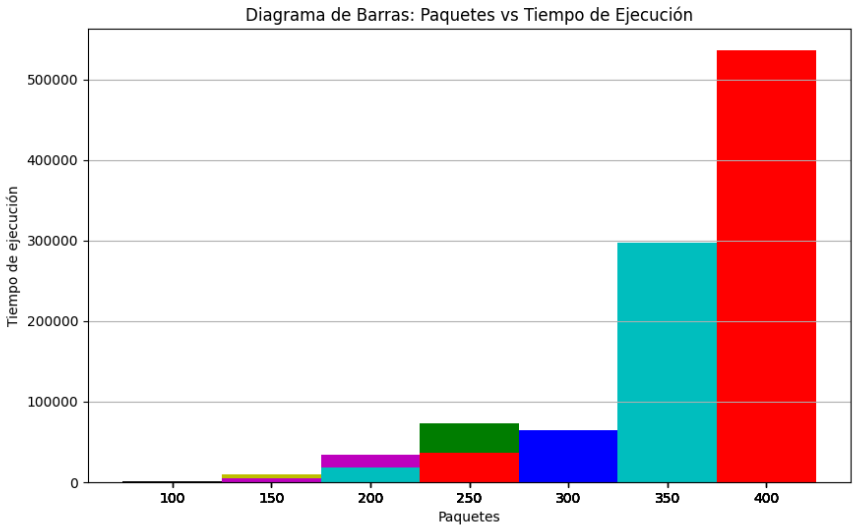
Planteamiento: Estudio de la evolución del tiempo de ejecución dependiendo de la magnitud.

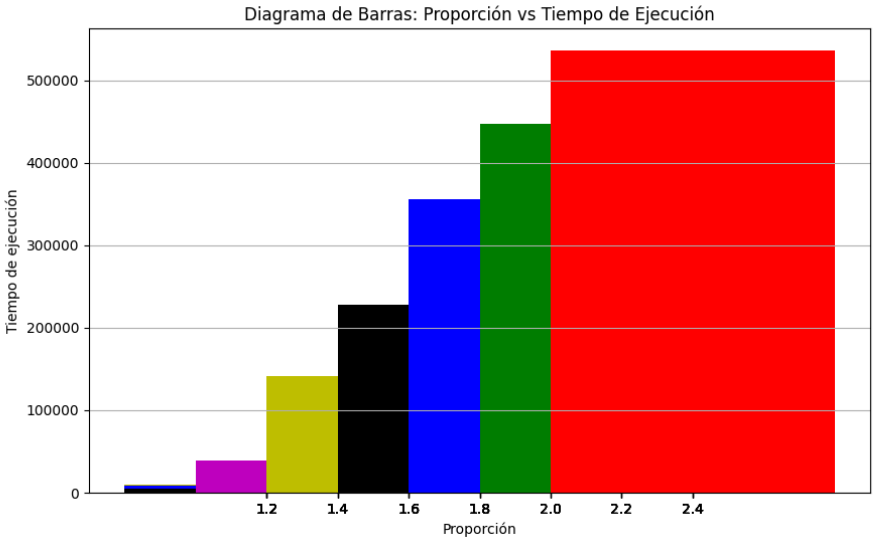
Hipótesis: Dado que Hill Climbing debe calcular todos los sucesores posibles, creemos que un mayor número de paquetes hará que el tiempo de ejecución incremente exponencialmente, mientras que la proporción incrementará el tiempo *polinómicamente*.

Metodología:

- Primero, fijamos 100 paquetes y aumentamos la proporción 1,2 de 0,2 en 0,2.
- Segundo, fijamos una proporción de 1,2 y aumentamos el número de paquetes 100 de 50 en 50.
- Se usa el algoritmo Hill Climbing.
- Ejecutamos el programa 1 única vez con cada combinación de valores, debido al gran valor que alcanzan.

Resultados:





Conclusiones:

Podemos observar cómo el tiempo de ejecución en los paquetes aumenta de manera cuadrática, lo que implica que, a medida que aumentan los valores de entrada o la complejidad del problema, el tiempo requerido para procesar esos paquetes se incrementa a un ritmo que se puede describir con una parábola. Sin embargo, es interesante notar que el costo asociado a esta proporción parece aumentar de forma mucho más rápida.

Para confirmar nuestra hipótesis, podemos realizar un análisis más detallado. Al examinar las métricas, se evidencia que, en general, estamos en lo correcto al evaluar la evolución del tiempo de ejecución. Esta tendencia sugiere que, a medida que el sistema se enfrenta a una mayor carga o a configuraciones más complejas, el tiempo de procesamiento no solo se ve afectado de manera predecible, sino que también está acompañado de un incremento significativo en los costos operativos. Esto nos invita a reflexionar sobre la eficiencia del sistema y la necesidad de optimizar ciertos procesos para manejar adecuadamente estos aumentos en el tiempo y en los costos.

2.5. Quinto experimento:

Este experimento pretende descubrir si el número de ofertas es relevante para mejorar el heurístico de los costes. ¿Qué comportamiento tiene el heurístico respecto a la proporción de las ofertas? ¿Merece la pena aumentar el número de ofertas?

Observación: Una mayor proporción aumenta el valor de la solución.

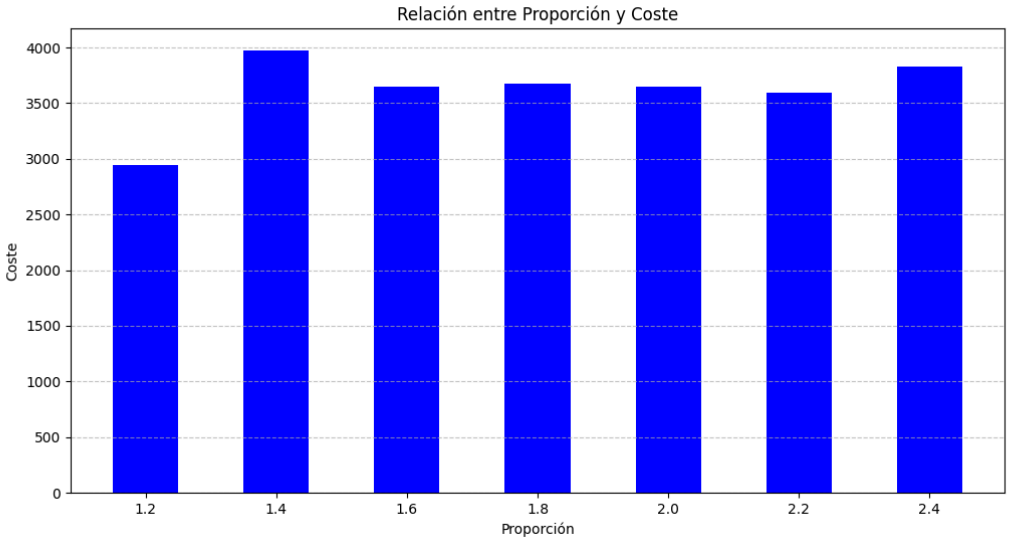
Planteamiento: Estudio de la importancia de la cantidad/ proporción de las ofertas.

Hipótesis: Afectará más a la calidad de la solución la proporción que el número.

Metodología:

- Usamos los resultados del experimento anterior donde observamos el comportamiento del heurístico según la proporción de las ofertas.

Resultados:



Conclusiones:

Nos damos cuenta de que la hipótesis era incorrecta ya que la proporción no parece afectar de manera significativa al coste. Aumentar el número de ofertas no parece afectar demasiado al coste total del heurístico.

2.6. Sexto experimento:

Este experimento evalúa cómo afecta la felicidad de los usuarios, calculada en función de la rapidez de entrega, sobre los costes de transporte y almacenamiento. Se explorará cómo varían estos costes y el tiempo de ejecución al ajustar la ponderación de la felicidad en la función heurística del algoritmo Hill Climbing. Para realizarlo recogeremos muestras con 5 valores de porcentajes distintos con nos resultaran en el **“valor ponderación Heurístico”.**

Observación:

La inclusión de la felicidad de los usuarios como factor en la función heurística puede tener efectos diferentes sobre los costes de transporte y almacenamiento. Es posible que la ponderación asignada a la felicidad impacte en el comportamiento del algoritmo, mejorando o empeorando estos costes según el peso relativo dado a este factor.

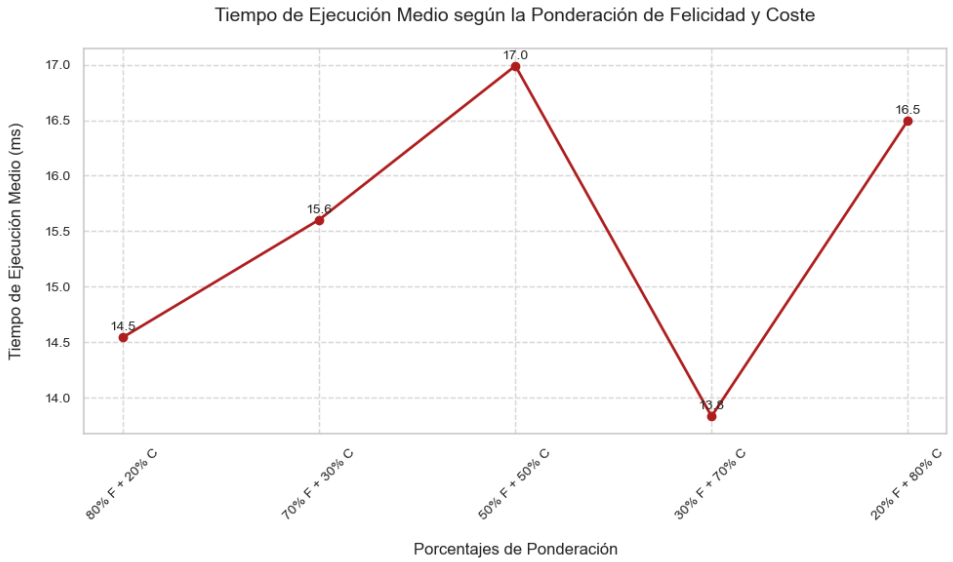
Planteamiento: Comparar cómo varían los costes de transporte y almacenamiento y el tiempo de ejecución al ajustar la ponderación de la felicidad de los usuarios en la función heurística del algoritmo Hill Climbing.

Hipótesis: Aumentar la ponderación de la felicidad reducirá los costes de transporte y almacenamiento al promover entregas más rápidas, aunque un exceso de importancia en este factor podría aumentar el tiempo de ejecución sin mejorar significativamente los costes.

Metodología:

- Se usa el algoritmo de Hill Climbing ajustando la ponderación de la felicidad en la función heurística.
- Ejecutamos el programa 100 veces con semillas de generación aleatorias, 100 paquetes y asignación inicial aleatoria (comprobada en el apartado 1 que es la que da mejores resultados).
- Comparamos los costes de transporte, almacenamiento y tiempo de ejecución para distintas ponderaciones de la felicidad.
- Observamos tendencias para identificar el rango de ponderación que optimiza la relación entre felicidad y costes.
- Partimos de que habíamos trabajado con el valor del coste en negativo para tratar de minimizarlo, pero que en la gráfica lo tomamos en positivo para que sea más visual la progresión positiva del valor total de la ponderación.

Resultados:



Conclusiones:

Coste:

Al aumentar la ponderación de la felicidad, el coste total de transporte y almacenamiento se incrementa de manera significativa. La configuración con mayor peso en el coste (20% felicidad + 80% coste) resulta en el menor coste, mientras que la configuración inversa (80% felicidad + 20% coste) muestra el coste más alto. Esto sugiere que priorizar en exceso la felicidad de los usuarios no es eficiente en términos de coste, y un enfoque más balanceado podría ser más óptimo.

Tiempo de Ejecución:

El tiempo de ejecución es menor en configuraciones donde el peso de la felicidad es moderado (como 30% felicidad + 70% coste), y aumenta tanto cuando la felicidad tiene un peso muy alto como cuando el coste es extremadamente prioritario. Esto indica que un equilibrio en la ponderación (cercano a 50% felicidad + 50% coste) ofrece un tiempo de ejecución más estable, evitando incrementos significativos en el procesamiento.

2.7. Séptimo experimento:

Este experimento evalúa cómo afecta la ponderación de la felicidad de los usuarios en los costes de transporte y almacenamiento utilizando el algoritmo **Simulated Annealing**, manteniendo la metodología y parámetros del experimento anterior con Hill Climbing.

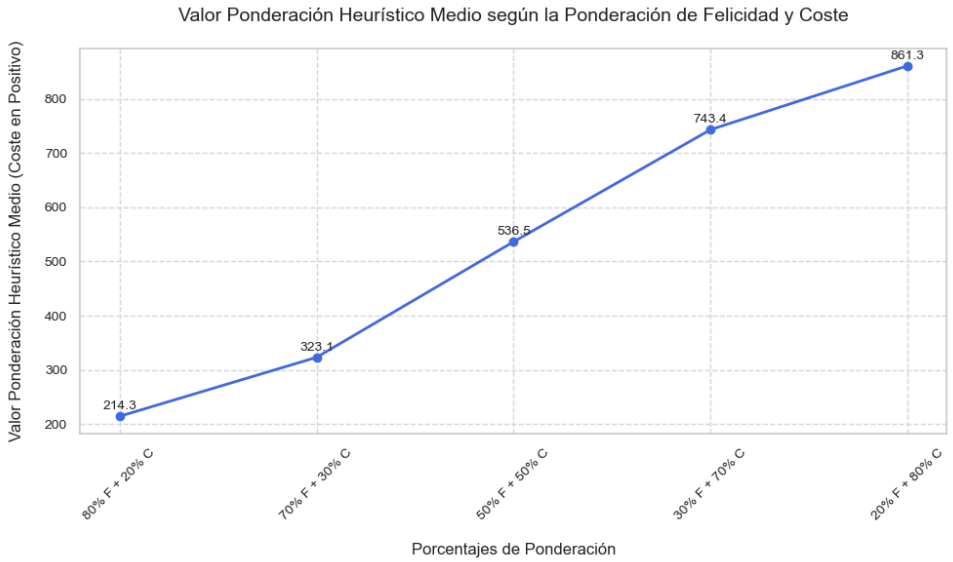
Observación:
La ponderación de la felicidad en la función heurística podría influir de forma diferente en los costes y tiempos de ejecución en Simulated Annealing, ya que este algoritmo tiene una exploración de soluciones más amplia que Hill Climbing.

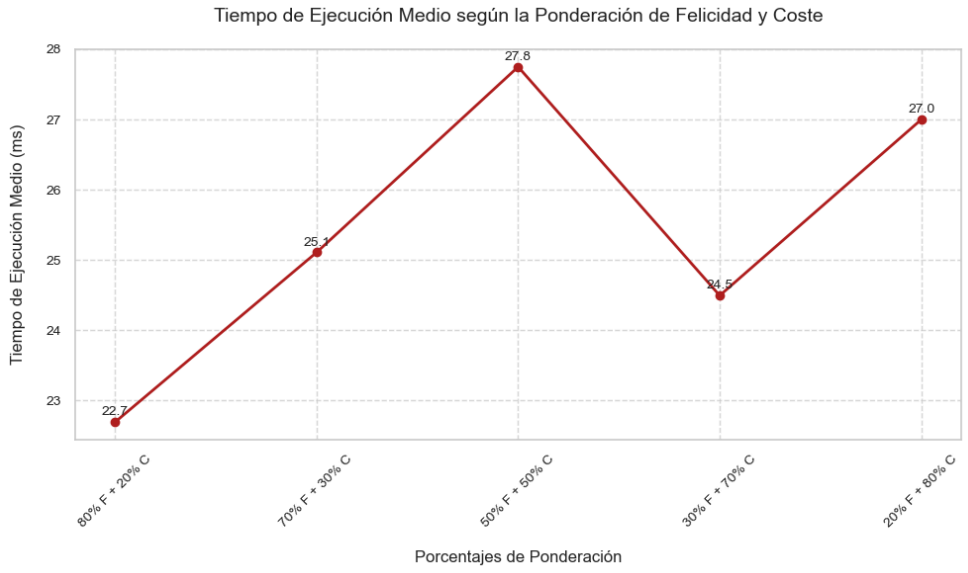
Planteamiento:
Comparar los costes de transporte, almacenamiento y tiempo de ejecución al ajustar la ponderación de la felicidad en la función heurística, ahora bajo el enfoque de Simulated Annealing.

Hipótesis:
Se mantiene la misma hipótesis planteada en el experimento anterior con Hill Climbing: aumentar la ponderación de la felicidad reducirá los costes al promover entregas rápidas, aunque un exceso de peso en la felicidad podría incrementar el tiempo de ejecución sin una mejora significativa en los costes.

Metodología:
Utilizamos Simulated Annealing con los mismos parámetros: 100 ejecuciones con semillas aleatorias, 100 paquetes y asignación inicial aleatoria. Comparamos los costes de transporte, almacenamiento y tiempo de ejecución para cinco ponderaciones distintas de la felicidad, buscando tendencias que optimicen la relación entre felicidad y costes.

Resultados:





Conclusiones:

Coste:

Al igual que en el experimento con Hill Climbing, observamos que el coste total aumenta al dar mayor ponderación a la felicidad. La configuración con menor ponderación en la felicidad (20% felicidad + 80% coste) muestra el menor coste, mientras que la configuración opuesta (80% felicidad + 20% coste) resulta en el coste más alto.

Esto confirma que un enfoque excesivo en la felicidad de los usuarios incrementa el coste, y que un balance intermedio ofrece una relación más eficiente entre coste y felicidad.

Tiempo de Ejecución:

El tiempo de ejecución promedio es menor en configuraciones balanceadas o con ponderación más alta en el coste, como en el caso de 30% felicidad + 70% coste. En configuraciones con alta ponderación en la felicidad, el tiempo de ejecución tiende a incrementarse. Esto sugiere que Simulated Annealing, al igual que Hill Climbing, es más eficiente cuando la ponderación está equilibrada o favorece levemente el coste. Por último se puede observar también que los valores promedio de tiempo de ejecución son mayores que en Hill Climbing, eso se debe a que este último no acota los nodos de expansión y en cambien en Simulated Annealing hemos acotado los pasos de expansión a 1000 nodos.

2.8. Octavo experimento:

Este experimento analiza cómo afectaría la solución si variamos el coste de almacenamiento, actualmente fijado en 0,25 euros por kilo diario. Sin realizar simulaciones, evaluamos los efectos de un aumento o disminución de este coste en la ponderación de felicidad y el coste de transporte.

Observación:
El coste de almacenamiento tiene un impacto directo en el coste total. Al modificar este valor, esperamos que el algoritmo ajuste la ponderación de felicidad y transporte para optimizar la solución.

Planteamiento:
Explorar teóricamente cómo cambiarían las soluciones al modificar el coste de almacenamiento, observando su influencia en la ponderación de felicidad y transporte en la función heurística.

Hipótesis:
Aumentar el coste de almacenamiento incentivará soluciones que prioricen la rapidez de entrega (mayor ponderación en felicidad), mientras que reducirlo permitirá que el algoritmo enfoque más en minimizar el coste de transporte.

Resultados:

Cambios en el Coste de Almacenamiento	Efecto en la Ponderación de Felicidad	Efecto en la Ponderación de Coste de Transporte
Aumento del coste de almacenamiento	Mayor ponderación en felicidad	Menor peso en coste de transporte
Reducción del coste de almacenamiento	Menor ponderación en felicidad	Mayor peso en coste de transporte

Conclusiones:

Al variar el coste de almacenamiento, el comportamiento del algoritmo cambia de acuerdo con la prioridad entre minimizar el tiempo de almacenamiento (felicidad) o el coste de transporte. Si el **coste de almacenamiento aumenta**, el algoritmo tenderá a preferir soluciones que minimicen el tiempo de almacenamiento, priorizando así la rapidez de entrega (mayor ponderación en felicidad). En cambio, si el **coste de almacenamiento disminuye**, el enfoque cambiará hacia la reducción del coste de transporte, asignando menor peso a la rapidez de entrega y, por tanto, a la felicidad de los usuarios.

3. Conclusiones

Realizar este trabajo nos ha dado una perspectiva mucho más profunda sobre el funcionamiento de los algoritmos Hill Climbing y Simulated Annealing, así como sobre los problemas de búsqueda local en general. Hemos visto cómo son herramientas extremadamente útiles para minimizar/maximizar una función objetivo, sobre todo cuando hay un espacio de soluciones posibles de gran tamaño.

Hemos comprobado cómo el algoritmo Simulated Annealing permite escapar máximos locales y obtener soluciones mejores que Hill Climbing una vez se calibran los parámetros adecuadamente. También hemos visto cómo, en problemas grandes, SA obtiene soluciones con bondad comparable a HC en una fracción del tiempo.

Durante la elaboración del trabajo hemos tenido la oportunidad de mejorar nuestros conocimientos sobre el software Java y Python y el IDE Intellij, así como habilidades como el trabajo coordinado en equipo y la elaboración de experimentos para analizar software.

Creemos que estos conocimientos podremos aplicarlos durante el resto del grado y de nuestra carrera profesional.

Por último, creemos que nuestro grupo se ha topado con varios problemas y hemos conseguido superarlos con creces. Hemos tenido un buen funcionamiento interno y estamos orgullosos del resultado final, tanto en programación como en documentación, y esperamos seguir trabajando juntos en otros trabajos de la asignatura.

4. Trabajo de innovación: Maia Chess

4.1. Tema seleccionado

Maia Chess es un innovador *bot* de ajedrez que se diferencia de los sistemas tradicionales al estar diseñado para replicar el estilo y los errores humanos. A través de la integración de técnicas avanzadas de IA, Maia Chess ha permitido a los jugadores analizar sus partidas de manera más profunda y personalizada, proporcionando una perspectiva única sobre su propio juego.

4.2. Responsabilidades del equipo

De momento, los avances con el proyecto de investigación se han centrado en la búsqueda de información relevante, verídica y útil. Por ello, todos hemos dedicado nuestros esfuerzos de la misma manera, y nos dividiremos las áreas de interés del proyecto más adelante.

4.3. Dificultades

De momento, la mayor dificultad ha sido encontrar detalles de la red neuronal que utilizan para entrenar al bot, y en encontrar fuentes de información fiables. Esto se debe a que es un proyecto universitario opensource.

4.4. Referencias

1. **Página web oficial de Maia Chess**

Maia Chess. *Maia Chess: An AI Designed to Match Human Play in Chess.* Consultada el 21 de octubre de 2024, [https://www.maiachess.com/].

- Descripción: Página web oficial del proyecto Maia Chess, una inteligencia artificial creada para jugar al ajedrez imitando las decisiones de los jugadores humanos en lugar de optimizar simplemente la victoria.

2. **Trabajo de investigación de Maia Chess**

McIlroy-Young, R., Sen, S., Kleinberg, J., & Anderson, A. *Maia: A Human-like Chess Engine*. Consultada el 29 de octubre de 2024, [https://arxiv.org/abs/2006.01855].

- Descripción: Artículo académico sobre Maia Chess, que detalla su desarrollo y objetivos.Publicado en arXiv, explica la tecnología y metodología utilizada para entrenar la IA en un estilo de juego humano.

3. **CSS Lab Blog: Maia Chess en KDD**

Maia Chess at KDD: Making an AI that Plays Like People. Consultado el 29 de octubre de 2024, [https://csslab.cs.toronto.edu/blog/2020/08/24/maia_chess_kdd/].

- Descripción: Blog de CSS Lab, describe la presentación de Maia en la conferencia KDD (Knowledge Discovery and Data Mining) 2020, donde los desarrolladores detallan el diseño y los logros del modelo Maia en la emulación de jugadas humanas.

4. **Microsoft Blog de Maia Chess**

Anderson, Ashton. *The Human Side of AI for Chess*. Consultado el 30 de octubre de 2024,[https://www.microsoft.com/en-us/research/blog/the-human-side-of-ai-for-chess/].

- Descripción: Publicación en el blog de investigación de Microsoft, discute el papel de Maia Chess en el avance de la IA para jugar de manera más humana y comprensible. Resalta el propósito del proyecto en el contexto de la IA ética y humana.

5. **Repositorio Maia Chess GitHub**

CSS Lab. *Maia Chess*. Consultada el 10 de noviembre de 2024, [https://github.com/CSSLab/maia-chess].

- Descripción: Repositorio oficial del código de Maia Chess en GitHub, proporcionado por el Computational Social Science Lab. Incluye el código fuente, instrucciones de uso y detalles técnicos del proyecto.