Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

PRÁCTICA 1

Algoritmos Básicos de la Inteligencia Artificial

Grau en intel·ligència artificial

Daniel Álvarez, Oriol Martí y Albert Roca

08/11/2024

Contents

1 Introducción

Este problema aborda la asignación óptima de paquetes a ofertas de transporte en un contexto en el que queremos maximizar. El objetivo es minimizar el coste total de transporte mientras se cumple con las prioridades de entrega de cada paquete, lo cual maximiza una métrica de "felicidad" o satisfacción del cliente. La estructura del problema incluye dos elementos principales:

- Paquetes: Cada paquete tiene un peso y una prioridad de entrega que representa la rapidez con la que debe llegar. Estas prioridades restringen las opciones de oferta a las que el paquete puede ser asignado.
- Ofertas: Cada oferta tiene una capacidad máxima de peso, un número de días de entrega y un coste por unidad de peso. Las ofertas de transporte con tiempos de entrega más largos suelen ser más económicas, pero pueden no cumplir con las prioridades de ciertos paquetes.

Estado del Problema
Un estado del problema se define como una asignación específica de cada paquete a una oferta. El estado incluye el coste acumulado de la asignación (calculado en función del peso de los paquetes y el coste de cada oferta) y la felicidad.

Espacio de Búsqueda y Operadores
El espacio de búsqueda está compuesto por todas las asignaciones posibles de paquetes a ofertas. Sin embargo, este espacio es limitado por las restricciones de capacidad y prioridad, lo que hace que solo ciertas combinaciones sean válidas. Los operadores usados para explorar este espacio son:

- Cambio de Oferta: Mueve un paquete de una oferta a otra que cumpla las restricciones.
- Intercambio de Paquetes: Intercambia las asignaciones de dos paquetes entre dos ofertas, siempre que ambas cumplan con los requisitos de capacidad y prioridad.

Justificación y Eficiencia de la Representación
Esta representación del problema es correcta, ya que permite evaluar fácilmente el coste y la felicidad de cada asignación, y los operadores permiten una exploración eficiente del espacio de búsqueda. Esto facilita la búsqueda de una solución óptima de menor coste mediante optimización local.

En conclusión, la estructura del problema y sus operadores permiten explorar el espacio de búsqueda de manera eficiente y encontrar configuraciones óptimas de asignación de paquetes a ofertas, optimizando el coste y, opcionalmente, la felicidad del cliente dentro de las restricciones logísticas.

El problema planteado en la práctica es un problema de búsqueda local, ya que el objetivo es encontrar una solución óptima o cercana al óptimo mediante ajustes incrementales en una solución existente, en lugar de construir una solución desde cero o explorar todas las posibles configuraciones. En este problema, cada estado representa una asignación específica de paquetes a ofertas. La solución se representa como un estado en el espacio de búsqueda, y cada estado es una posible asignación. La solución se va ajustando con base en modificaciones incrementales, moviéndonos de un estado solución a otro.

Por qué es un problema de búsqueda local: La búsqueda local explora soluciones cercanas al pasar de un estado a sus vecinos, como en este caso, mediante cambios de oferta o intercambios de paquetes. Estos movimientos permiten modificar asignaciones para encontrar configuraciones que optimicen una función objetivo de felicidad y coste.

En este problema, la búsqueda local es adecuada porque permite mejorar iterativamente la solución sin necesidad de evaluar todo el espacio de soluciones, que sería ineficiente dada su gran magnitud. En lugar de una búsqueda exhaustiva, se hacen pequeños ajustes en la asignación, evaluando si estos cambios mejoran la solución.

Sin embargo, la búsqueda local puede quedar atrapada en óptimos locales, por lo que técnicas como Simulated Annealing ayudan a escapar de estos y mejorar la solución. Por todas estas razones, este problema se ajusta bien a los métodos de búsqueda local, optimizando soluciones en espacios grandes sin requerir una exploración completa.

2 Estado del problema y representación

2.1 Clase StateRepresentation

Esta clase encapsula el estado actual de las asignaciones de paquetes para elementos. El estado incluye una lista de paquetes, una lista de ofertas, la asignación de cada paquete a una oferta.
Esta clase contiene las siguientes características:

- **Params**: Contiene los parámetros del problema (paquetes y ofertas) y los operadores.
- **Assignacions**: Representa la asignación actual de cada paquete a una oferta, lo que define el estado del sistema en un momento concreto.
- **Generate actions**: Genera las posibles acciones que se pueden realizar desde el estado actual, permitiendo explorar estados vecinos mediante dos tipos de operadores (cambio de oferta de un paquete o intercambio de paquetes entre ofertas (swap).
- **Apply action**: Aplica una de las estas acciones a un estado, generando un nuevo estado resultante.
- **Cost calcular**: Calcula el coste total de la asignación de paquetes a ofertas en un estado concreto.
- **Felicidad**: Calcula la felicidad de los clientes, basada en la diferencia que existía entre la prioridad de cada paquete y los días de entrega real de la oferta.

2.2 Espacio de estados

El espacio de estados es el conjunto de todas las posibles asignaciones de paquetes a ofertas, considerando todas las combinaciones posibles que cumplen las restricciones. Para pasar de un estado a otro usaremos los operadores, que serán explicados más adelante también. Un estado inicial se genera mediante la función generate initial state, la cual será explicada detalladamente en la siguiente sección de este informe, pero para tener una pequeña idea, filtra las ofertas según su precio, teniendo en cuenta si son ofertas prometedoras (precio por debajo de la media) u ofertas caras (precio por encima de la media).

2.3 Operadores

En nuestro caso tenemos únicamente dos operadores: El swap (intercambiar dos paquetes que están en dos ofertas diferentes) y el mover paquete (mover un paquete de una oferta a otra que tenga espacio disponible). Este operador tiene en cuenta que al realizar un cambio, se siga manteniendo la prioridad de paquetes y días. Antes de realizar una operación se comprueba que esta se pueda hacer.

2.4 Heurística

Más adelante en este informe comentaremos las tres heurísticas mencionadas y sus peculiaridades, y para qué se puede usar cada una. La heurística intentará guiar al algoritmo al camino óptimo en cada paso. Es una estimación de lo que queda para encontrar ese óptimo.

2.5 Tamaño del estado de búsqueda

El espacio de búsqueda incluye todas las posibles soluciones que cumplen con los requisitos del problema, es decir, todas las asignaciones de paquetes a ofertas que respetan el peso máximo permitido en cada oferta y cumplen con las prioridades de entrega. Dado un número de paquetes N y un número de ofertas M, el tamaño máximo del espacio de búsqueda sería M^N, en el caso ideal en que todas las combinaciones fueran válidas. Sin embargo, las restricciones de peso y prioridad reducen el número de combinaciones efectivas, limitando el espacio de búsqueda a las soluciones viables.

3 Elección y generación del estado inicial

Una buena solución inicial tiene que dar con el mínimo coste posible, la buena aproximación a la solución final que se quiere obtener. Eso quiere decir que buscamos una manera rápida de asignar los paquetes a ofertas y que tenga cierta relación con la solución final que queremos encontrar.

Para la creación de esta, el enfoque que hemos diseñado se basa en dividir las ofertas en dos categorías: ofertas baratas y caras. Calculando la media de los costes de las ofertas, obtendremos un valor a partir del cual podemos separar según si el precio de la oferta es mayor o menor que este. El coste de esta operación es lineal, por lo tanto, hemos hecho una segregación de precios sin tener que recurrir a métodos de ordenación, que tienden a ser más costosos.

La asignación de paquetes se realiza de forma ordenada, comprobando la asignabilidad en las ofertas baratas antes que en las caras. Este proceso prioriza las asignaciones baratas siempre que sea posible, manteniendo una estructura simple y efectiva para la asignación inicial. El único inconveniente es para los paquetes con prioridad de entrega, que tienen poca probabilidad de ser asignables en las ofertas baratas, pero van a tener que iterar sobre ellas. Si es necesario según la experimentación, este va a ser un punto de mira para una mejora

```
def crear_solucio_inicial_baratescares(l_paquetes,l_ofertas):
    llista_preus = [y.precio for y in l_ofertas]
    accum = 0
    for i in llista_preus:
        accum += i
    mitjana = accum/len(llista_preus)
    oferta_por_paquete = [0] * len(l_paquetes)
    peso_por_oferta = [0.0] * len(l_ofertas)
    i = 0
    ofertas_baratas = []
    ofertas_caras = []
    while i<len(l_ofertas):
        if llista_preus[i]<= mitjana:
            ofertas_baratas.append(i)
        else:
            ofertas_caras.append(i)
        i+=1
    barates_i_cares = [ofertas_baratas,ofertas_caras]
    for id_paquete in range(len(l_paquetes)):
        paquete_asignado = False
        oferta_potencial = 0
        while not paquete_asignado:
            for barata_o_cara in barates_i_cares:
                for id_oferta in barata_o_cara:
                    if asignable_estricta(l_paquetes[id_paquete], l_ofertas[id_oferta]):
                        oferta_potencial = id_oferta
                        if l_paquetes[id_paquete].peso + peso_por_oferta[oferta_potencial] \
                        <= l_ofertas[oferta_potencial].pesomax:
                            peso_por_oferta[oferta_potencial] = peso_por_oferta[oferta_potencial] \
                                                                + l_paquetes[id_paquete].peso
                            oferta_por_paquete[id_paquete] = oferta_potencial
                            paquete_asignado = True
                            break
                if paquete_asignado:
```

```
                    break
            if not paquete_asignado:
                print("no s'ha trobat oferta per al ",id_paquete, l_paquetes[id_paquete].peso)
                break
    return oferta_por_paquete
```

Por otra parte, hemos creado una asignación que la llamamos 'ordenada', que simplemente itera sobre las ofertas siguiendo su orden natural, y basada en las restricciones de prioridad y en la capacidad de peso de cada oferta, asigna cada paquete a la primera lista disponible que se encuentra.

```
def crear_asignacion_ordenada(l_paquetes, l_ofertas):
    # Función para crear una asignación de paquetes a ofertas
    # basada en la prioridad de cada paquete y la capacidad de cada oferta.
    print()
    def es_asignable(paquete, oferta):
        # Verifica que la prioridad del paquete coincida con los días de la oferta
        return not ((paquete.prioridad != 0 or oferta.dias != 1)
                    and (paquete.prioridad != 1 or oferta.dias != 2)
                    and (paquete.prioridad != 1 or oferta.dias != 3)
                    and (paquete.prioridad != 2 or oferta.dias != 4)
                    and (paquete.prioridad != 2 or oferta.dias != 5))

    oferta_por_paquete = [-1] * len(l_paquetes)
    peso_por_oferta = [0.0] * len(l_ofertas)

    # Recorrer cada paquete
    for id_paquete, paquete in enumerate(l_paquetes):
        for id_oferta, oferta in enumerate(l_ofertas):
            # Comprueba asignabilidad
            if es_asignable(paquete, oferta):
                # Verifica si el peso
                if paquete.peso + peso_por_oferta[id_oferta] <= oferta.pesomax:
                    oferta_por_paquete[id_paquete] = id_oferta
                    peso_por_oferta[id_oferta] += paquete.peso
                    break  # Sale del bucle
        else:
            print(f"No se encontró oferta para Paq= {id_paquete} dentro de las restricciones.")
    return oferta_por_paquete
```

Y por último, como un generador de estado inicial hemos usado el de la generación subóptima del archivo abia_azamon.py que se nos entregaba ya vía el Racó. Hemos usado este para poder tener un tercer generador i poder tener un abanico mayor de asignaciones para realizar una comparación exigente.

La complejidad de todas las funciones es de $n\_p \times n\_o$ aunque la asignación de barates_cares tiene el coste extra de iterar sobre las ofertas previamente para hacer la separación.

Basados en los resultados de la experimentación, se ha determinado crear_solucio_inicial_baratescares como la mejor solución de las tres, por lo tanto, esta va a ser la asignación inicial para todos los experimentos futuros.

4 Representación y análisis de los operadores

Los operadores son una parte importante de la búsqueda, su correcta aplicación permite realizar una búsqueda completa, además de ser rápida y sin costes innecesarios. Desde el principio teníamos una cosa clara: un operador tenía que ser MovePaquete, el cual cambia la oferta a la que está asignado un paquete. Este sencillo operador puede hacer por sí mismo una búsqueda muy completa, y tiene un coste algorítmico no demasiado grande.

Operador Move:

- **Condiciones de aplicabilidad:** Mueve el paquete p a la oferta o si la prioridad del paquete lo permite y si no se supera el peso máximo de transporte una vez añadido el paquete.
- **Factor de ramificación:** Dado el número de paquetes N y el número de ofertas Ø, el factor de ramificación es (N $\times O^2$) $Este fragmento de código simplificado representa la ramificación del operador$

```
for oferta\_j in range(len(self.params.l\_ofertas)):
    for p\_i in range(len(self.params.l\_paquetes)):
        if self.assignacions[p\_i] == oferta\_j:for oferta\_k in range(len(self.params.l\_ofertas)):
            \#(CODI OMÈS)Condició: contenidor diferent i té espai lliure suficient i es assignableyield
            MoverOferta(p\_i, oferta\_j, oferta\_k)
```

Aún así, la búsqueda no es tan completa como nos gustaría. En algún punto, es perfectamente posible que no pueda cambiar de manera rápida y obvia un paquete de oferta, ya que la oferta destino ya está llena, y se deben ejecutar múltiples operaciones de move. Para eso nos puede servir un operador que intercambie la oferta de dos paquetes simultaneamente. A este operador le llamaremos Swap.

Operador Swap:

- **Condiciones de aplicabilidad:** Intercambia las asignaciones de los paquetes p1 y p2 si después de la permutación de cada paquete se satisfacen las prioridades de entrega y las sumas de los pesos de transporte no se exceden debido a este cambio
- **Factor de ramificación:** Si tenemos N paquetes; el factor de ramificación será N^2 . Este fragmento de codigo representa su ramificación:

```
for oferta\_j in range(len(self.params.l\_ofertas)):
    for p\_i in range(len(self.params.l\_paquetes)):
        if self.assignacions[p\_i] == oferta\_j:for oferta\_k in range(len(self.params.l\_ofertas)):
            \#(CODI OMÈS)Condició: contenidor diferent i té espai lliure suficient i es assignableyield
            MoverOferta(p\_i, oferta\_j, oferta\_k)
```

En nuestro primer experimento, vimos que tener swap reduce el precio, pero no de manera óptima. Es decir, para lo poco que reduce el precio, el tiempo prácticamente se duplica. Había que seguir experimentando, pero al menos había dos cosas claras. 1.El operador move tiene plaza asegurada y 2.Hay que intentar reducir el coste algorítmico de Swap, ya que la idea es efectiva pero demasiado costosa (coste N^2)

Simplificando mínimamente la función swap para que no itere para todos los paquetes dos veces, hemos diseñado dos pequeñas variaciones.

- **Alternativa Swap1:** Se realiza un único cambio. En la segunda iteración, se itera entre range(i+1,len(l_paquetes)). De esta forma conseguimos reducir el coste algorítmico notablemente, siendo el coste medio de la segunda iteración de $(N-1)/2$ i por tanto su factor de ramificación $\frac{N\times(N-1)}{2}$
- **Alternativa Swap2:** En vez de iterar una segunda vez sobre los paquetes, se escoge un paquete aleatorio y se prueba si hay posibilidad de ejecutar el intercambio de paquetes. El coste es lineal ya que solo iteramos una vez sobre los paquetes, y luego se genera un valor aleatorio. Factor de ramificación, N

En esta segunda experimentación los resultados fueron favorables. Vimos disminuir el tiempo de ejecución de manera muy eficaz, y finalmente escogimos la primera alternativa de Swap1, la de factor de ramificación $\frac{N\times(N-1)}{2}$. [1]

A partir de aquí pensamos en probar algún otro operador, pero no tuvimos una idea que sobre el papel pudiese ser más efectiva que los operadores que teníamos. Decidimos seguir adelante con los siguientes apartados de la práctica, siempre atentos a si encontrábamos una nueva idea para añadir otro operador.

[1] Para ver los resultados de los experimentos, consultar apartado 7.1.

5 Análisis de la función heurística

La heurística es sin duda una de las partes más importantes del proyecto, ya que es la que va a guiar al algoritmo para poder hacer una búsqueda correcta y obtener el mejor resultado. Buscamos optimizar el coste de transporte, que se calcula a partir del peso que hay en cada oferta y su precio por kilogramo. Esto significa que no nos aumenta precio el usar muchas ofertas. Por tanto, de primeras no va a tener mucho sentido plantear la heurística a partir de la ocupación en cada oferta para usar el mínimo de ofertas. La solución irá guiada principalmente por el coste de transporte, el cual depende de la ocupación de los paquetes más baratos y más caros, y del tiempo que se quede cada paquete en el almacén. Las dos heurísticas que planteamos son:

- **HEURÍSTICA 1:** Breve y sencilla, esta heurística va a ser el precio de coste de la asignación de paquetes en cuestión, dado por la función calcular_coste()

```
    def cost_calcular(self) -> float:
        cost = 0.0
        for paquete, oferta in enumerate(self.assignacions):
            cost += self.params.l_paquetes[paquete].peso * self.params.l_ofertas[oferta].precio
            if self.params.l_ofertas[oferta].dias in [3, 4]:
                cost += 0.25 * self.params.l_paquetes[paquete].peso
            elif self.params.l_ofertas[oferta].dias == 5:
                cost += 0.5 * self.params.l_paquetes[paquete].peso
        return cost
```

- **HEURÍSTICA 2:** En esta heurística introduciremos el concepto de la ocupación mediante la entropía, enfocado a maximizar la ocupación en los paquetes baratos, y a minimizar la de los caros siempre que sea posible.

```
#calcular free_spaces
#dividir ofertas en baratas y caras
total\_entropy = 0
for oferta, libre in enumerate(free\_spaces):
    occupancy = 1 - (libre / self.params.l\_ofertas[oferta].pesomax)
    if occupancy > 0:
        if oferta in ofertas\_baratas:
            total\_entropy = total\_entropy - (occupancy * math.log(occupancy))return total\_entropy
```

Para encontrar la mejor heurística hemos querido visualizar el valor del coste total en cada paso de búsqueda del algoritmo, para ver si se está llegando a minimizar este valor. No sería de extrañar que el complejo diseño de la heurística por entropía no sea muy favorable en comparación con la que valora simplemente el coste.

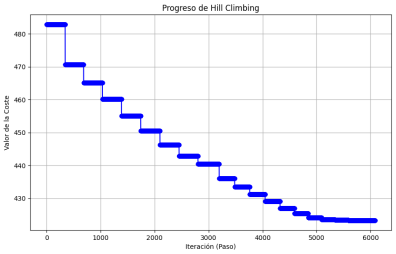
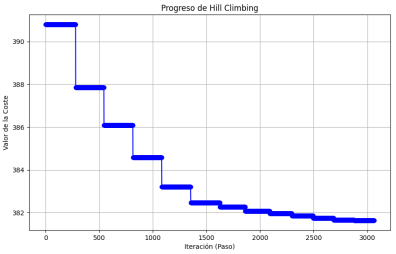
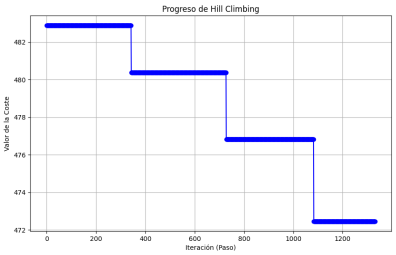
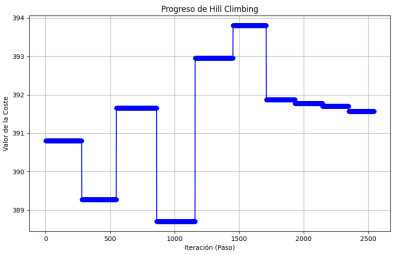
	Semilla 33	**Semilla 69**
Heurística 1 Coste		
Heurística 2 Entropía		

Table 1: Comparación de heurísticas con diferentes semillas

Ejecutando con las mismas dos semillas el programa, cambiando únicamente la heurística aplicada, vimos que la del coste era altamente eficaz, mientras que la de la entropía era muy mala, llegando incluso a aumentar el coste. Con estas dos primeras ejecuciones nos bastó para determinarlo, e incluso provando otras semillas los resultados eran similares, así que solo hemos considerado incluir estos dos ejemplos. No quisimos seguir diseñando una heurística aún más complicada si esta ya no había dado un resultado positivo, y la más simple era tan buena.

Por lo tanto, la heurística que vamos a usar será Heurística 1, que valora el coste.

6 Experimentación

En este apartado del proyecto vamos a reportar como se han elaborado las experimentaciones para evaluar el funcionamiento de la búsqueda y poder optimizarla. En el primer experimento vamos a tratar de manera más extensa lo que hemos tenido en cuenta para hacer el experimento, así como el método aplicado, considerando que en el resto de experimentos no se profundizara tanto.

6.1 Experimento 1: Influencia de los operadores

En este experimento queremos ver cuál es la combinación de los operadores que nos minimiza el coste. Tal y como hemos mencionado ya en diversas ocasiones en este informe, los operadores son SwapPaquete y MoverPaquete. A partir de los siguientes gráficos comentaremos cuál de las combinaciones cumple nuestras premisas.

Observación: Tiene que haber una combinación de operadores que pueda encontrar la mejor solución posible.

Planteamiento: Se generarán 10 semillas aleatorias para hacer 10 ejecuciones con valores distintos entre sí de cada combinación de operadores. Estas 10 semillas no cambian entre cada experimento. En total se ejecutan 4 experimentos (sin operadores, con move, con swap, y con move+swap)

Hipótesis:

- H0: Con todas las combinaciones de operadores obtenemos los mismos resultados.
- H1: Existe una combinación de operadores con la que obtenemos mejores resultados.

Implementación y resultados:

	Sense			Move			Swap			Move i Swap		
semilla	costes	iteracions	temps	costes	iteracions	temps	costes	iteracions	temps	costes	iteracions	temps
0	465.115	1	0.000972	414.27	20	0.259744	457.975	4	0.007982	412.29	23	0.392195
1	398.29	1	0.000541	366.94	17	0.230868	387.55	6	0.042039	361.415	21	0.416671
2	502.745	1	0.000683	483.245	13	0.11368	487.28	7	0.030051	472.2	17	0.230983
3	368.605	1	0.00068	354.265	9	0.093179	354.74	8	0.053274	348.14	14	0.254563
4	535.12	1	0.000966	509.14	18	0.176529	524.425	8	0.032576	499.695	27	0.492025
5	549.855	1	0.000735	497.325	16	0.127359	543.08	5	0.034287	492.92	19	0.312097
6	517.735	1	0.000821	462.66	23	0.325524	500.295	5	0.022092	456.03	24	0.430629
7	464.505	1	0.000629	438.885	12	0.100522	460.93	2	0.003339	434.875	13	0.150149
8	464.655	1	0.00044	410.695	18	0.210387	443.625	8	0.034118	406.155	21	0.361235
9	457.16	1	0.000594	416.975	16	0.178861	449.76	4	0.023768	408.79	22	0.457954
average	472.3785	1	0.000706	435.44	16.2	0.181665	460.966	5.7	0.028352	429.251	20.1	0.34985

Figure 1: Tabla Coste,Iteración,Tiempo para cada operador

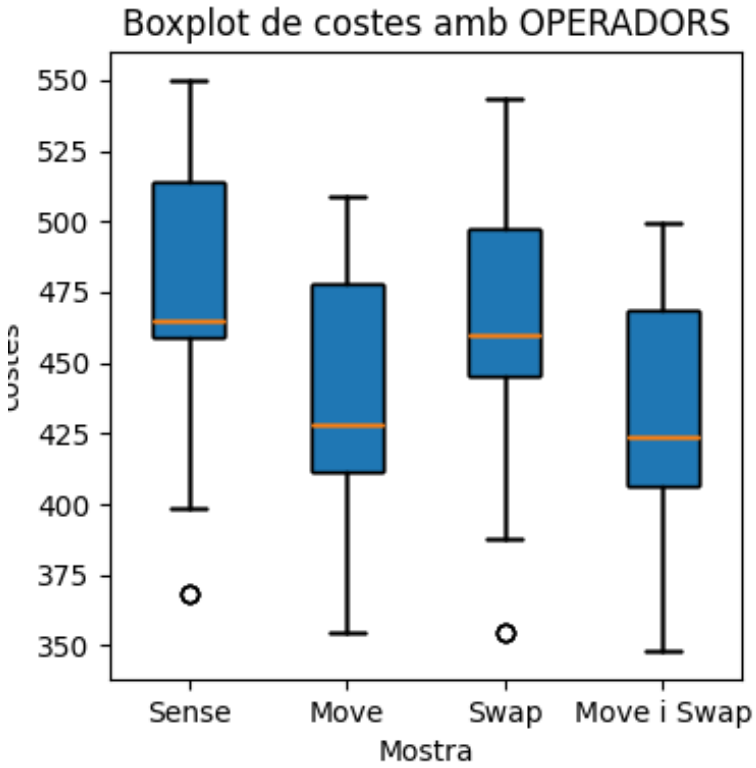


Figure 2: Boxplot coste operadores

Observando estos dos gráficos sobre los valores tomados en la experimentación del coste y del tiempo de ejecución, inmediatamente se ve que el operador que más efecto tiene en realizar una búsqueda es el move. Cuando no se aplica este operador, el tiempo de ejecución es prácticamente 0 y no se llega a minimizar el coste. La primera afirmación que podemos hacer, es que el uso de move es indiscutible. Y respeto al uso de swap+move, mirando el coste exacto en la tabla de valores, se ve que su uso es mejor que usar solo move, haciéndolo un operador válido para la búsqueda. No obstante, hay que valorar que su mejora de precio es casi inapreciable en el barplot de coste, pero, en cambio, el barplot de tiempo de ejecución, prácticamente se duplica.

Esta observación nos da indicios de que swap tarda demasiado en ejecutarse, aunque vale la pena usarlo, ya que nos reduce el coste, el principal objetivo del proyecto. Por este motivo, realizaremos una segunda experimentación, haciendo dos pequeñas variaciones en la lógica del swap y reduciendo su factor de ramificación, como ya se ha comentado previamente en apartados anteriores. Se repetirá el metodo usado en el previo experimento, aunque esta vez solo vamos a cambiar la función swap a usar (sin usar swap, swap original, variación swap1, variación swap2).

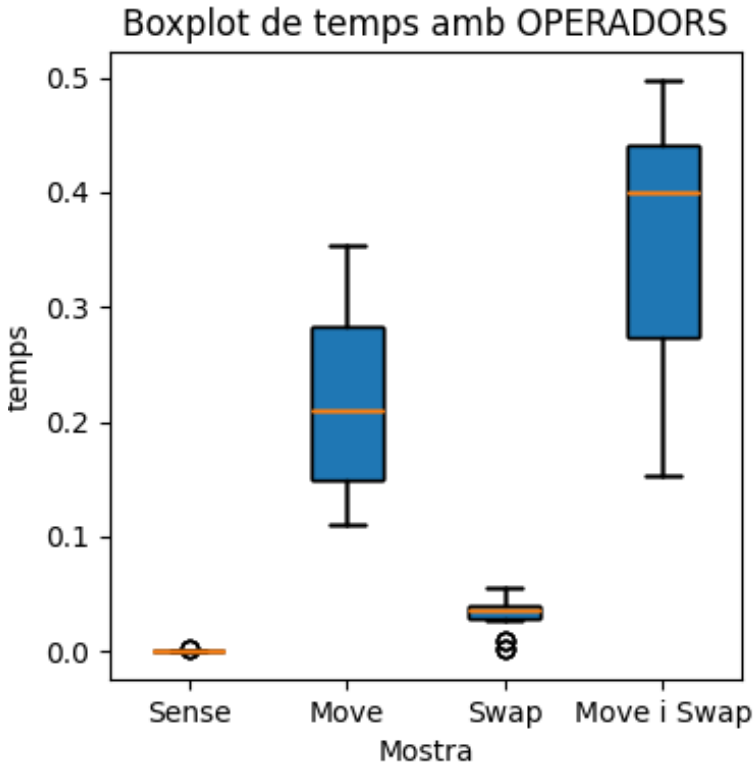


Figure 3: Boxplot tiempo operadores

	No Swap			Swap			Swap1			Swap2		
semilla	costes	iteracions	temps	costes	iteracions	temps	costes	iteracions	temps	costes	iteracions	temps
0	414.27	20	0.348277	412.29	23	0.412719	412.29	23	0.415067	412.815	22	0.386263
1	366.94	17	0.271952	361.415	21	0.706847	361.415	21	0.429198	363.19	18	0.280663
2	483.245	13	0.141794	472.2	17	0.270548	472.2	17	0.286434	475.085	17	0.227331
3	354.265	9	0.105858	348.14	14	0.281342	348.14	14	0.310437	350.995	12	0.129223
4	509.14	18	0.201509	499.695	27	0.557269	499.695	27	0.529933	499.325	25	0.288561
5	497.325	16	0.141013	492.92	19	0.382271	492.92	19	0.238917	494.335	18	0.153847
6	462.66	23	0.326272	456.03	24	0.504944	456.03	24	0.442823	457.67	24	0.395712
7	438.885	12	0.135874	434.875	13	0.17608	434.875	13	0.267349	438.53	13	0.138333
8	410.695	18	0.247042	406.155	21	0.404908	406.155	21	0.419208	408.39	22	0.401421
9	416.975	16	0.204632	408.79	22	0.46497	408.79	22	0.689382	413.565	18	0.248611
average	435.44	16.2	0.212422	429.251	20.1	0.41619	429.251	20.1	0.402875	431.39	18.9	0.264997

Figure 4: Tabla Coste,Iteración,Tiempo para cada Swap

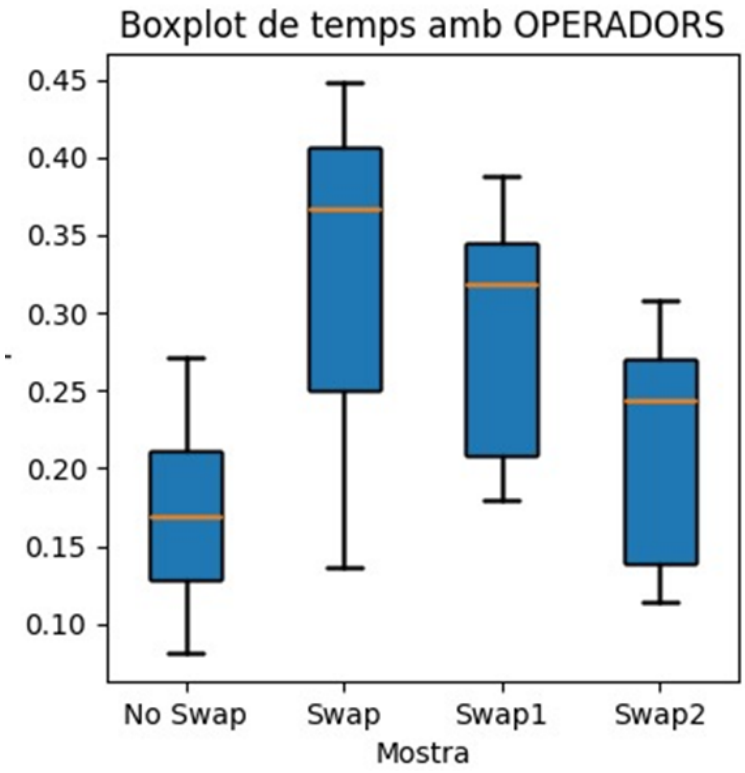


Figure 5: Boxplot tiempos Swap

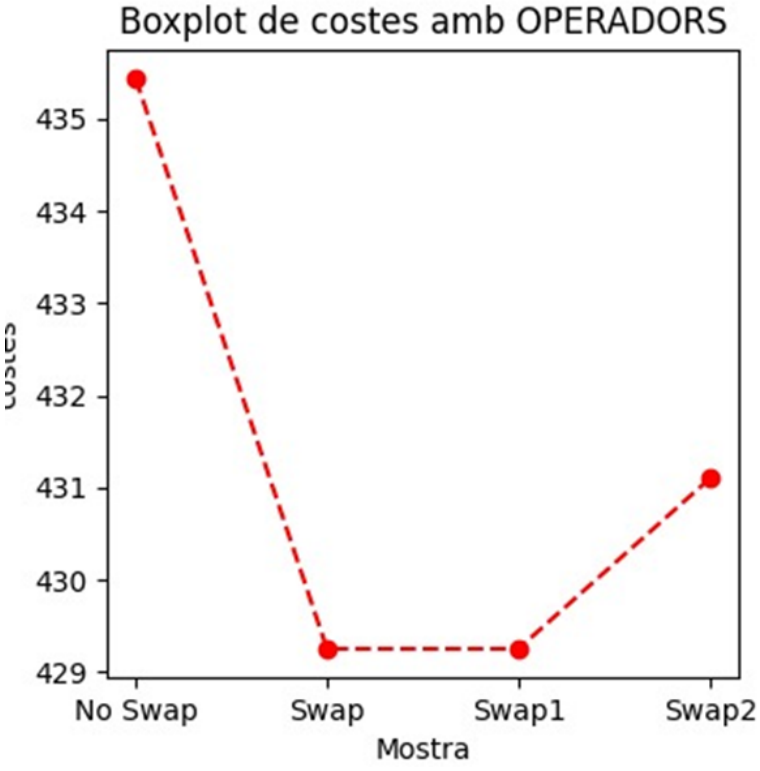


Figure 6: Enter Caption

Si buscásemos minimizar el tiempo de ejecución, Swap2 es la opción para escoger. No obstante, la prioridad es reducir el coste, y vemos da un coste más elevado. Por otro lado, Swap1 mantiene los mismos costes que el Swap original, y además tarda menos tiempo. Es por eso que nos quedaremos con este ejemplo, ya que hemos conseguido reducir la complejidad algorítmica y el tiempo de ejecución sin repercutir en el resultado.

6.2 Experimento 2: Influencia de Solución inicial

Observación: determinar qué estrategia de creación de solución inicial funciona mejor para la función heurísticat utilizada en la sección anterior con el mismo escenario. Usando Hill Climbing.

Planteamiento: Generamos soluciones con ambos generadores y comparamos sus resultados. **Hipótesis:**

- H0: Con todos los generadores obtenemos la misma solución
- H1: Uno de ellos genera mejores soluciones.

Implementación y resultados: Para poder decidir cuál es la mejor repetiremos el procediemiento del ejercicio anterior, haciendo diez experimentos aleatorios comparando el coste y tiempo de ejecución para cada generador de solución inicial. La tabla se vería así:

Métrica	**Crear Solución Inicial (Barates/Caras)**	**Asignación Subóptima**	**Asignación Ordenada**
Precio Inicial	488.2795	495.6230	511.7605
Precio Post-Búsqueda	449.8285	451.4790	452.1530
Tiempo Ejecución Solución Inicial (s)	0.000471	0.000259	0.000484
Tiempo Ejecución Total (s)	0.3008	0.3551	0.4541

Table 2: Comparación de Asignaciones: Precios y Tiempos de Ejecución

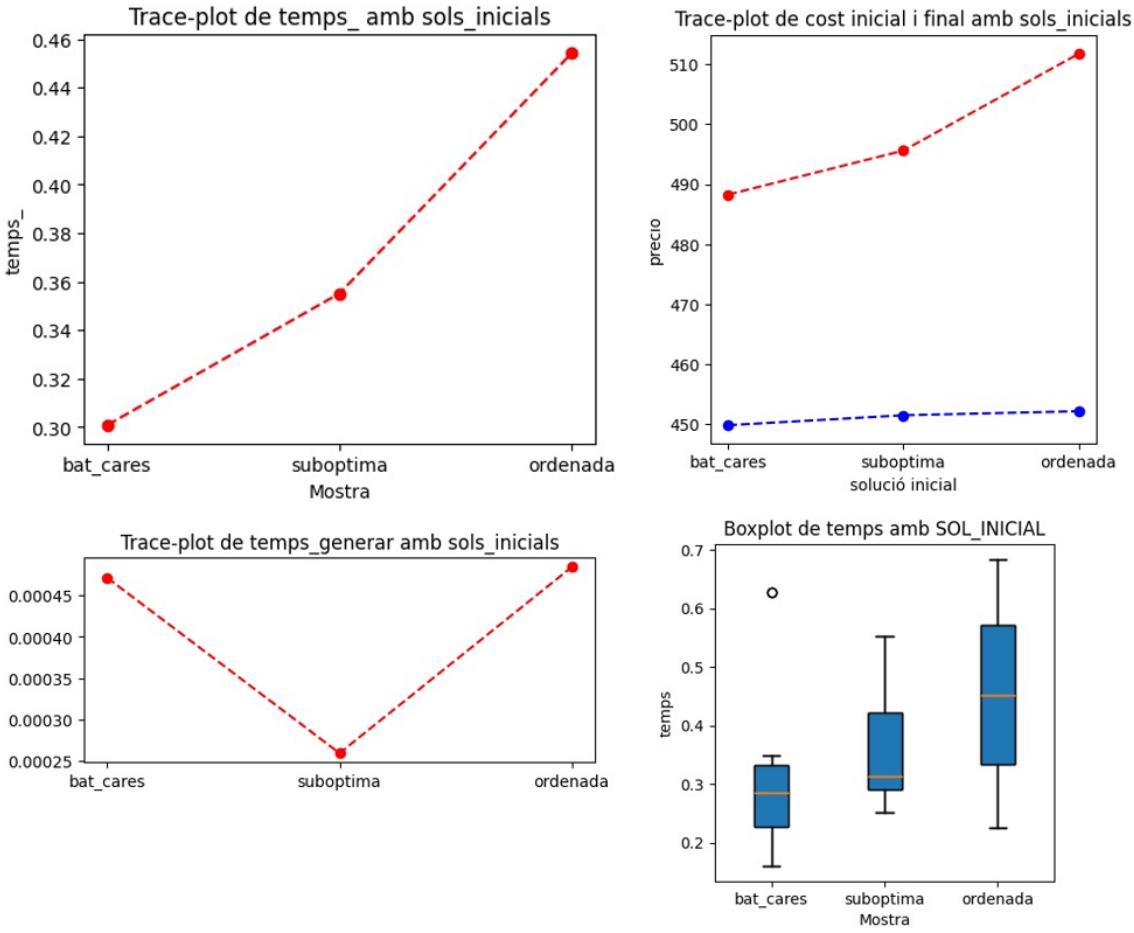


Figure 7: Comparación de estrategias de solución inicial en tiempo de ejecución y coste para las opciones Barates_Cares, Subóptima y Ordenada.

A partir de estos valores, de primeras vemos que la solución inicial que divide las ofertas entre baratas i caras (con label bat_caras) es la que obtiene mejores resultados de obtención de coste. A nivel de tiempo, si bien el hecho de calcular la mediana y crear dos sublistas aumenta el tiempo de generar la solución inicial en comparación con la asignación_suboptima(aleatoria), este valor representa una ínfima parte del tiempo de ejecución total. El tener una asignación centrada en reducir costes, se llega a una mejor solución con mucho menos tiempo de búsqueda que las otras dos, haciéndola la mejor opción como solución inicial.

Otra cosa que queríamos comprobar, era el riesgo de no encontrar una oferta para un paquete en la asignación inicial. Era un problema que vimos ocasionalmente durante la experimentación en la asignación subóptima, y queríamos exactamente ver cuál fallaba más, es por eso que experimentamos a ver cuál era la tasa en la que cada algoritmo tiende a no asignar más a menudo, la tabla es la siguiente:

N Paquetes	**Asignación Subóptima**	**Asignación Barates_Cares**
50	0.021	0.005
60	0.044	0.004
70	0.054	0.004
80	0.041	0.002
90	0.040	0.000
100	0.045	0.000

Table 3: Missrate por tipo de asignación y número de paquetes

Como se puede observar, la generación subóptima tiende a fallar hasta cuatro veces más que la asignación de baratas y caras. Esto significa que es notablemente más eficiente en cuanto a la reducción de paquetes mal asignados en comparación con la Asignación Subóptima, especialmente a medida que el número de paquetes aumenta. Con esto concluímos que el primer método es una opción más robusta y precisa para el enfoque de nuestro trabajo.

6.3 Experimento 3: Determinar los parámetros del Simulated Annealing

El objetivo de este experimento es poder determinar los tres parámetros que nos den un mejor resultado de nuestra búsqueda local usando el simulated annealing.
Para este experimento implementaremos el conjunto de los dos operadores establecidos previamente y la heurística que minimiza el coste de la empresa. La estrategia para determinar los parámetros óptimos ha sido la misma que se ha implementado en el documento del enunciado; haciendo una experimentación exhaustiva usando un número de paquetes igual a cincuenta e implementando la semilla 1234.

Observación:

Tiene que haber una combinación de los parámetros de la función que calcula el Simulated Annealing que de mejores resultados que otra combinación

Planteamiento:

Generamos soluciones con diversos parámetros para ir poco a poco acotando sus valores.

Hipótesis:

- H0: Con todas las combinaciones de parámetros obtenemos los mismos resultados.
- H1: Existe una combinación de parámetros con la que obtenemos mejores resultados.

Implementación y resultados:

En cuanto a la experimentación, hemos usado valores de k = {1,5,25,125} y Lambda = {1,0.01,0.001}, para empezar, hemos establecido un número alto de iteraciones para una vez tener los valores de k y λ poder ajustarlos. Así se vería la tabla de los resultados:

COSTE SA	**K**			
λ	**1**	**5**	**25**	**125**
1	473.735	470.050	**468.349**	**468.459**
0.01	**468.115**	476.854	488.560	478.300
0.001	**468.689**	488.295	478.100	487.304

Cómo podemos ver, hay varios valores que son bastante similares en cuánto al coste final, estos son los marcados en negrita, así que para poder precisar más en este aspecto cogeremos los parámetros de cada uno de los resultados y ejecutaremos cinco experimentos para cada combinación de parámetros. El que obtenga una media de coste más baja será el 'ganador'. En la siguiente tabla se muestran los resultados de esto.

	K = 25 / λ=1	**K = 1 / λ=0.01**	**K = 1 / λ=0.001**	**K = 125 / λ=1**
1-	468.349	468.115	468.689	469.459
2-	466.635	467.534	467.990	479.495
3-	477.875	471.114	471.974	479.530
4-	467.860	471.825	470.129	471.804
5-	470.195	470.470	478.150	474.879
PROMEDIO	**470.183**	**469.812**	**471.386**	**475.033**

Cómo se puede ver, la combinación de parámetros que da un resultado más constante es la de K = 1 y λ=0.01, debido a que su coste promedio es menor. A continuación incluímos unos gráficos de ejecución del Simulated Annealing para cada combinación de parámetros.

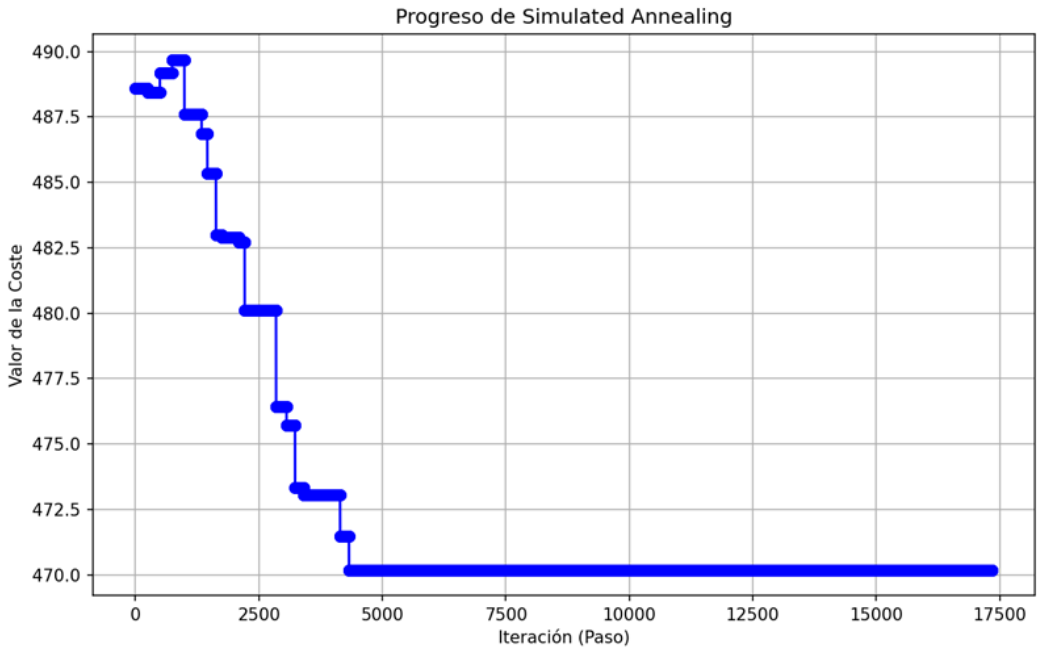


Figure 8: Ejecución del SA para k = 1, λ = 25

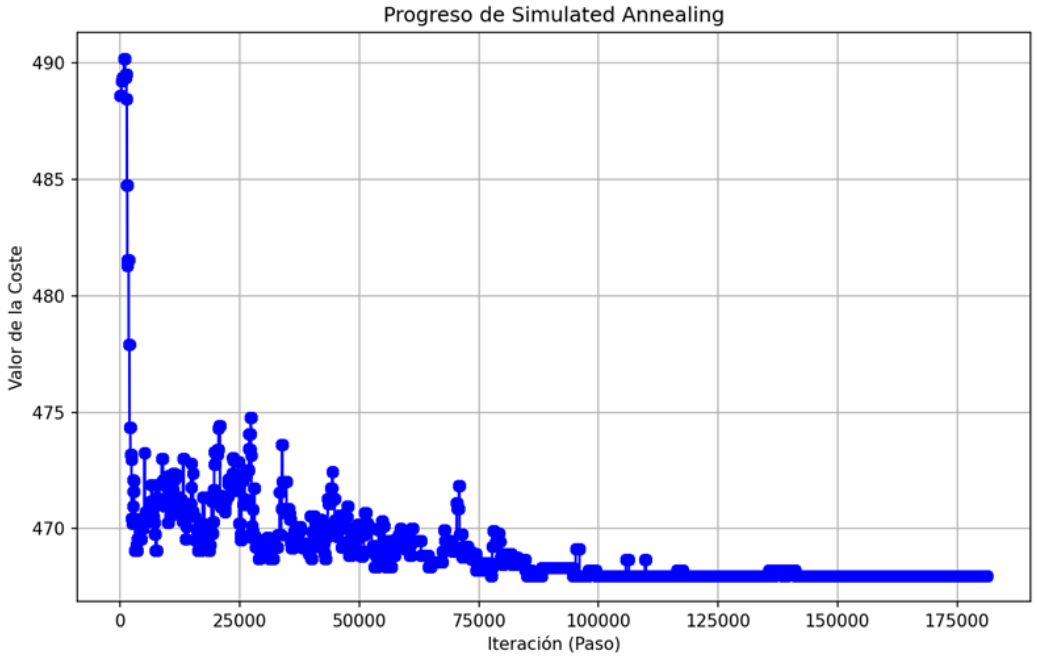


Figure 9: Ejecución del SA para k = 1, λ = 0.001

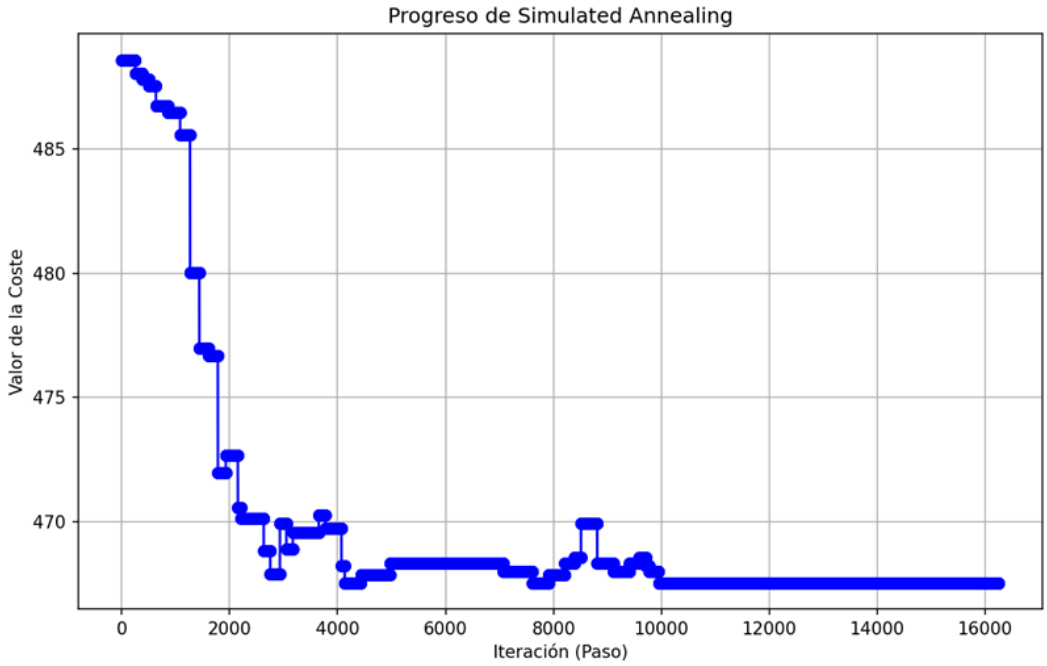


Figure 10: Ejecución del SA para k = 1, $\lambda = 0.01$

Finalmente, hemos reducido el número de iteraciones a 200, debido a que con este número ya nos converge a la mejor solución. Para poder deducir este número, hemos realizado distintas pruebas con estos parámetros de k y λ en distintas semillas fijando un número de paquetes igual a 50.

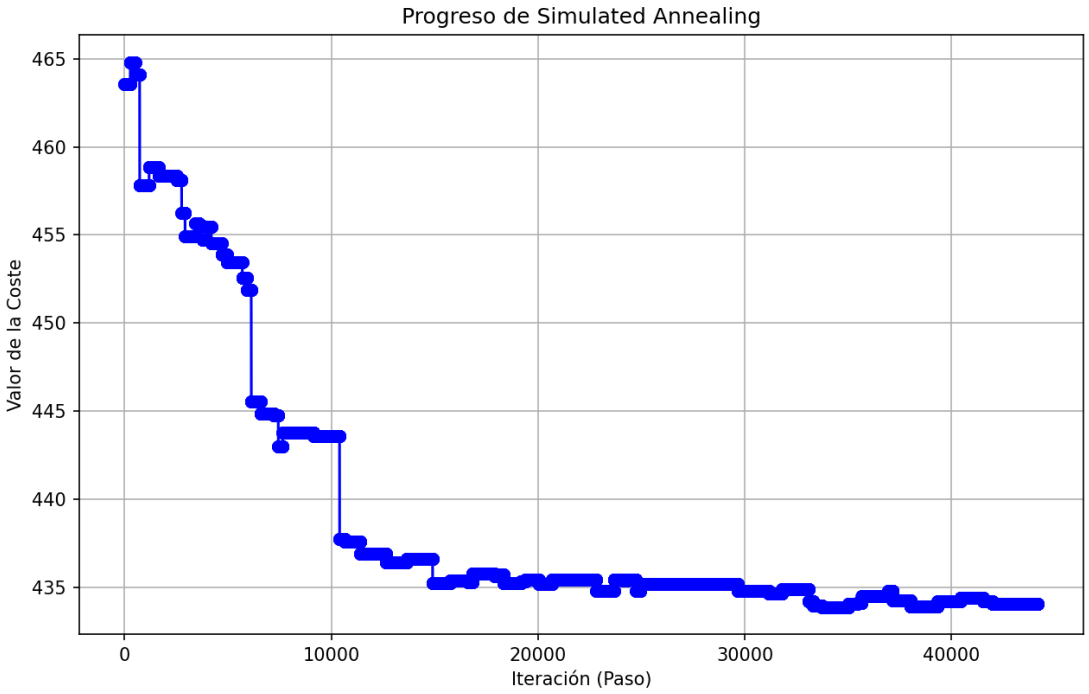


Figure 11: SA en la semilla 4561

6.4 Experimento 4: Evolución del tiempo de ejecución

La herística elegida para este experimento es la que no tiene en cuenta la felicidad (la cuál será hablada más adelante en este informe). Esta heurística solo intenta minimizar el coste.
La cantidad de paquetes elegida va de 50 a 150 (para realizar la comparación con la diferente cantidad de paquetes). El primer gráfico realizado es el siguiente:

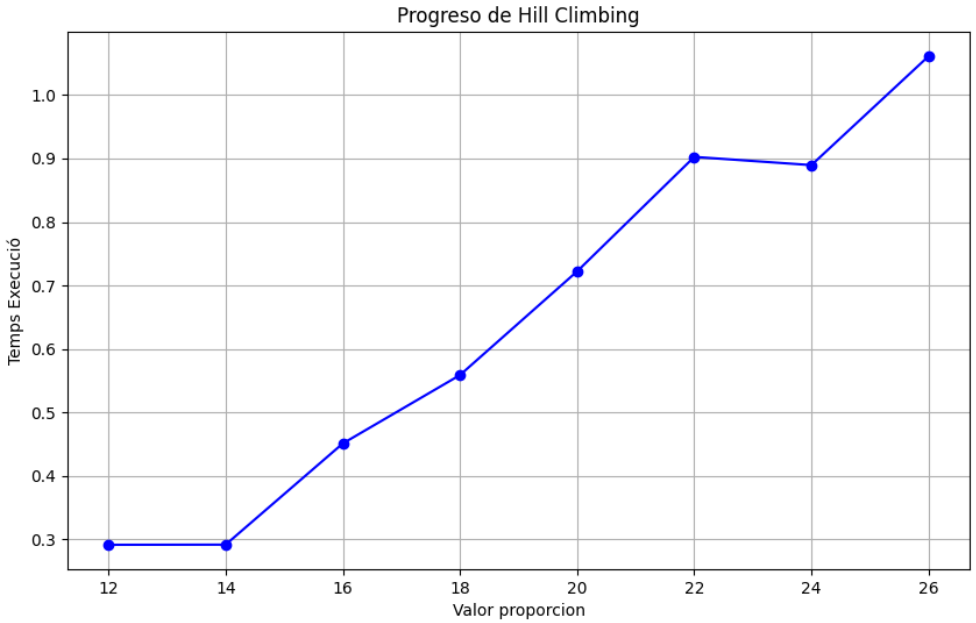


Figure 12: Evolución del tiempo de ejecución en función de la proporción

6.4.1 Análisis del primer gráfico

El gráfico muestra una tendencia creciente en el tiempo de ejecución a medida que incrementamos la proporción entre la capacidad de las ofertas y el peso total de los paquetes. Esto implica que, conforme la capacidad de las ofertas se hace mayor en relación con el peso total de los paquetes, el algoritmo de Hill Climbing necesita más tiempo para encontrar una solución óptima o cercana al óptimo. Este comportamiento sugiere que un mayor espacio de soluciones incrementa la complejidad del problema desde la perspectiva del algoritmo, dado que ahora tiene más configuraciones de asignación posibles que explorar.
Con una proporción baja (empezando en 1.2), el espacio de búsqueda es más restringido, ya que las ofertas tienen una capacidad limitada y ajustada en relación al peso de los paquetes. Esto significa que solo hay unas pocas combinaciones viables de asignación, lo cual reduce el número de vecinos que el algoritmo tiene que considerar en cada iteración, facilitando encontrar soluciones válidas rápidamente. En contraste, a medida que aumentábamos la proporción (en nuestro caso hasta 2.6), las ofertas tienen una capacidad considerablemente mayor en relación al peso de los paquetes. Esto genera muchas más posibilidades de asignación, ya que las restricciones de capacidad son menos es-

trictas. El algoritmo tiene que explorar un espacio de búsqueda más amplio, evaluando más estados y, por lo tanto, requiere más tiempo.

En este gráfico observamos que el tiempo de ejecución no crece de forma estrictamente lineal con el aumento de la proporción. Entre algunos valores, como por ejemplo entre 2.0 y 2.4, el tiempo de ejecución se estabiliza, lo que sugiere que el algoritmo encuentra una especie de "punto de equilibrio" temporal en el que el aumento en el espacio de búsqueda no implica una mayor dificultad de exploración. Esto podría deberse a que, a partir de cierta capacidad, la mayoría de las combinaciones de asignación se vuelven viables, por lo que el algoritmo no encuentra tantos obstáculos en forma de restricciones. Sin embargo, al aumentar aún más la proporción hasta 2.6, el tiempo de ejecución vuelve a subir, lo que podría indicar que el exceso de opciones disponibles dificulta encontrar la asignación óptima, ya que el algoritmo tiene que explorar más combinaciones y el factor de ramificación es mayor.

En el contexto de diseño de problemas o configuración de sistemas, mantener la proporción en un rango moderado (por ejemplo, alrededor de 1.2 a 1.6) podría ser ideal. En este rango, el problema es lo suficientemente complejo para garantizar que el algoritmo explore varias soluciones, pero no tan amplio como para hacer que el tiempo de ejecución sea excesivo.

Este gráfico muestra que existe una relación no lineal entre la proporción de capacidad y el tiempo de ejecución del algoritmo de Hill Climbing. En un rango de proporción bajo, el problema es más simple, lo que permite tiempos de ejecución más rápidos debido al espacio de búsqueda restringido. A medida que la proporción aumenta, el problema se vuelve más complejo debido al mayor número de combinaciones de asignación, lo cual se refleja en tiempos de ejecución más largos. Para minimizar el tiempo de ejecución y evitar una sobrecarga en el algoritmo, sería prudente mantener la proporción en un rango moderado. Esto permite un equilibrio entre la flexibilidad en la asignación de paquetes y la eficiencia computacional del algoritmo.

6.4.2 Análisis del segundo gráfico

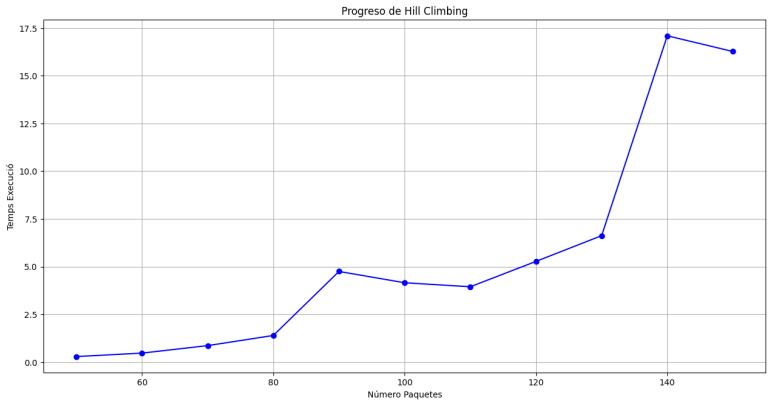


Figure 13: Evolución del tiempo de ejecución en función del número paquetes

Se observa una clara tendencia ascendente en el tiempo de ejecución conforme se incrementa el número de paquetes. Esto tiene sentido, ya que un mayor número de paquetes implica un espacio de

búsqueda más grande y una mayor complejidad en la asignación de paquetes a las ofertas disponibles.

Entre los 70 y 120 paquetes, el tiempo de ejecución presenta un crecimiento más moderado y en algunos puntos incluso parece estabilizarse o disminuir ligeramente. Esto nos podría indicar que el algoritmo de Hill Climbing encuentra configuraciones cercanas al óptimo con relativa rapidez antes de alcanzar el límite superior del espacio de búsqueda.

A partir de los 120 paquetes, el tiempo de ejecución comienza a aumentar de forma más pronunciada, alcanzando un pico entre 140 y 150 paquetes. Este comportamiento sugiere que, al aumentar el número de paquetes, el algoritmo comienza a encontrar más desafíos en la búsqueda de una asignación eficiente, lo que implica más evaluaciones y movimientos para alcanzar una solución.

En términos de complejidad, esto indica una relación no lineal entre el número de paquetes y el tiempo de ejecución. La complejidad del problema crece de manera significativa a medida que los paquetes aumentan, lo que hace que el tiempo de ejecución se dispare cuando se alcanza un volumen considerable de paquetes.

6.4.3 Análisis del tercer gráfico

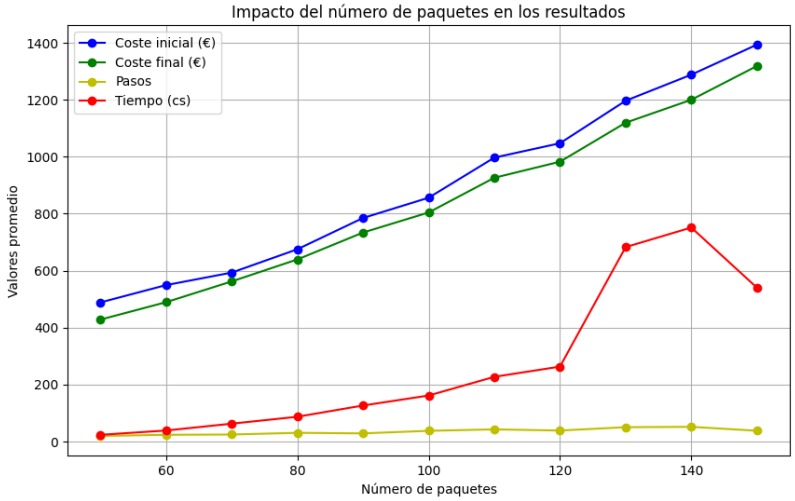


Figure 14: Evolución del tiempo de ejecución en función del número paquetes

Viendo este último gráfico, se puede determinar que aumentando la proporción de ofertas respecto a los paquetes, el coste inicial incrementa, debido a que se asignan paquetes a opciones de mayor precio en la configuración inicial. Aunque el algoritmo optimiza este coste en la solución final, el beneficio de reducción se estabiliza a medida que la proporción crece.
El número de pasos del algoritmo se mantiene constante, pero el tiempo de ejecución aumenta significativamente con la proporción, lo que indica que manejar más ofertas incrementa la complejidad y el tiempo de procesamiento.

6.5 Experimento 5: Análisis

En este apartado se nos pregunta cómo se comporta el coste de transporte y almacenamiento analizando los resultados del experimento en el que se aumenta la proporción de ofertas, y si merece la pena aumentar el número de ofertas. Viendo los gráficos del apartado anterior, podemos determinar lo siguiente:

En un primer momento, a medida que incrementa el número de ofertas, el coste inicial presenta un aumento gradual. Esto se debe a que, al haber más opciones de transporte y almacenamiento, algunos paquetes se asignan a opciones más costosas en las soluciones iniciales.

Sin embargo, incrementar el número de ofertas tiene beneficios limitados en términos de reducción de costes tras la optimización, ya que el coste final tiende a estabilizarse. Además, el incremento en el tiempo de ejecución sugiere que, más allá de un cierto punto, agregar más ofertas puede no ser eficiente, ya que se incurre en un coste computacional adicional sin obtener una mejora significativa en el coste final. Por lo cual, incrementar la proporción de ofertas puede ser útil hasta un cierto punto.

6.6 Experimento 6: Cómo afecta la felicidad

Los parámetros para experimentar sobre cómo afecta la felicidad eran los siguientes:

1. **Rango de ponderación de felicidad mínima:** Representa el valor inicial de la ponderación que se dará a la felicidad en el experimento. Este parámetro define el punto de partida para valorar qué tan importante es la felicidad en la función objetivo.

2. **Rango de ponderación de felicidad máxima:** Representa el valor máximo de la ponderación de la felicidad en el experimento. Este parámetro define el límite superior hasta el cual se incrementará la importancia de la felicidad en la función objetivo.

3. **Aumento en la ponderación de felicidad por experimentación:** Es el incremento en la ponderación de felicidad entre cada experimento. Define en que valor se incrementa la ponderación desde el valor mínimo hasta el máximo, permitiendo evaluar cómo cambia el resultado al aumentar progresivamente la importancia de la felicidad.

4. **Número de experimentos por cada aumento en la ponderación:** Indica cuántas veces se repetirá el experimento para cada valor de ponderación de felicidad. Esto permite obtener un promedio y reducir el impacto de la aleatoriedad en los resultados.

5. **Números de paquetes por experimento:** Representa la cantidad de paquetes que se incluirán en cada experimento. Este valor afecta la complejidad del problema y permite ver cómo la asignación de un número determinado de paquetes influye en los resultados.

Tras realizar diferentes experimentos, cambiando los parámetros, obseramos que a medida que aumentábamos la felicidad, el gráfico bajaba y los valores eran muy similares, y no podíamos observar con claridad cuál era la ponderación de felicidad correcta. Es por eso que hicimos el siguiente gráfico:

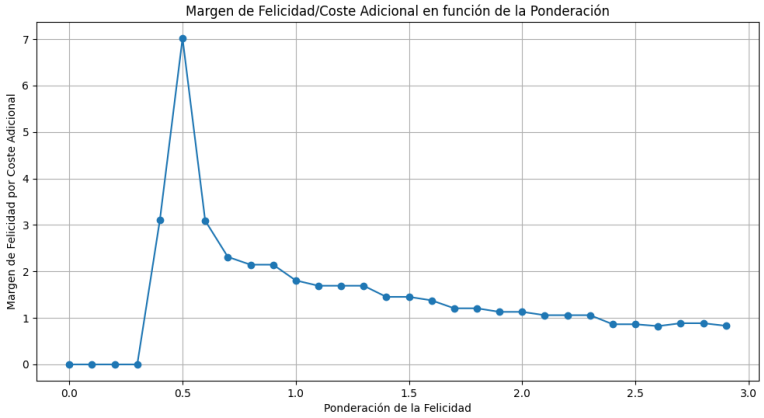


Figure 15: Primer experimento con un rango de ponderación de felicidad máxima = 3

El margen indica cuánto aumenta la felicidad en relación al coste adicional que se incurre al incrementar la ponderación de la felicidad. En otras palabras, muestra la eficiencia en términos de felicidad

obtenida por cada unidad de coste adicional a medida que ajustas la ponderación. Todo es en base a la primera solución (ponderación de felicidad = 0).
Tal y cómo se observa. La tendencia del margen de felicidad/coste a medida que aumentamos la ponderación de la felicidad, tiende a bajar. Pudimos observar claramente que el valor que maximiza el margen es con ponderación de felicidad = 0.5. Probamos numerosas combinaciones de paquetes y ponderaciones, y en todos los valores eran iguales a los de los gráficos presentados a continuación:

6.6.1 Interpretación de los experimentos realizados:

- **Equilibrio Ideal**: En este gráfico, el punto de equilibrio se observa en una ponderación de felicidad alrededor de 0.5. En esta región, el margen de incremento en felicidad por cada unidad de coste adicional es máximo, alcanzando un valor destacado. Esto indica que este es el punto donde el incremento en felicidad es más eficiente en relación al coste. En términos prácticos, esta ponderación ofrece el mejor balance entre felicidad y coste, donde el gasto adicional resulta claramente justificado por el aumento en felicidad.

- **Rango de Oportunidad Moderada**: A medida que la ponderación de felicidad aumenta, en el rango entre 0.75 y 1.5, el margen de felicidad por coste adicional se estabiliza, aunque disminuye progresivamente. En este rango, aún es posible obtener beneficios adicionales en felicidad, pero a costa de una mayor inversión. Aunque la eficiencia en coste es menor en comparación con el equilibrio ideal, este rango sigue siendo razonable para aquellos que buscan priorizar la felicidad sin que el coste se dispare. Este rango es ideal para aquellos que desean una felicidad más elevada y están dispuestos a asumir un incremento moderado en el coste.

- **Desperdicio de Coste en Ponderaciones Altas**: A partir de una ponderación de felicidad superior a 1.5, el gráfico muestra que el margen de felicidad por coste adicional disminuye de manera pronunciada y se mantiene bajo. En este rango, los incrementos en felicidad requieren un gasto considerablemente mayor, pero producen un beneficio marginal en felicidad. Esta es una zona donde el coste deja de estar justificado en relación con el beneficio en felicidad, ya que cada incremento en felicidad se obtiene con una inversión significativa.

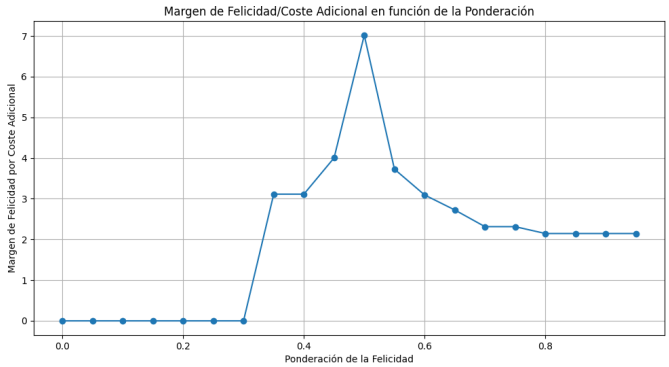


Figure 16: Primer experimento con un rango de ponderación de felicidad máxima = 1

6.7 Experimento 7: Felicidad y Simulated Annealing:

6.7.1 Implementación

Observación: Generar diferentes soluciones ponderando (utilizando diferentes valores) el valor de felicidad de la heurística

Hipótesis:

- H0: El costo del transporte y almacenamiento tendrán valores similares.
- H1: El precio aumentará o disminuirá considerablemente

Implementación y resultados: En este apartado el objetivo es repetir el proceso implementado en el experimento 6, el cuál introducía los parámetros de la felicidad en la heurística. Los parámetros que se usarán serán los mismos; rango de ponderación de felicidad mínima y máxima, aumento en la ponderación de felicidad por experimentación, número de experimentos por cada aumento en la ponderación y números de paquetes por experimento. Tras realizar diferentes experimentos, cambiando los parámetros, usando 50 paquetes, obseramos una tendencia. Aquí se expresan los distintos resultados en una tabla y gráfico:

ponderacion_felicidad	coste_promedio	felicidad_promedio	margen_felicidad_coste
0.0	429.1850	0.0	0.000000
0.5	431.8100	4.5	1.714286
1.0	437.4575	12.0	1.450589
1.5	438.1050	12.5	1.401345
2.0	451.5575	20.0	0.893955
2.5	456.9125	23.0	0.829501
3.0	469.4800	28.0	0.694875
3.5	473.1275	27.5	0.625818
4.0	484.9075	30.5	0.547355
4.5	493.5200	32.0	0.497396
5.0	493.8825	33.0	0.510066
5.5	500.6850	34.5	0.482517
6.0	511.8300	36.0	0.435598
6.5	512.5375	36.5	0.437899
7.0	521.6050	37.0	0.400346
7.5	527.3300	37.5	0.382088
8.0	566.5675	43.0	0.312995
8.5	554.5675	41.5	0.339987
9.0	561.5000	43.5	0.328761

Table 4: Tabla de ponderación de felicidad, coste promedio, felicidad promedio y margen de felicidad-coste.

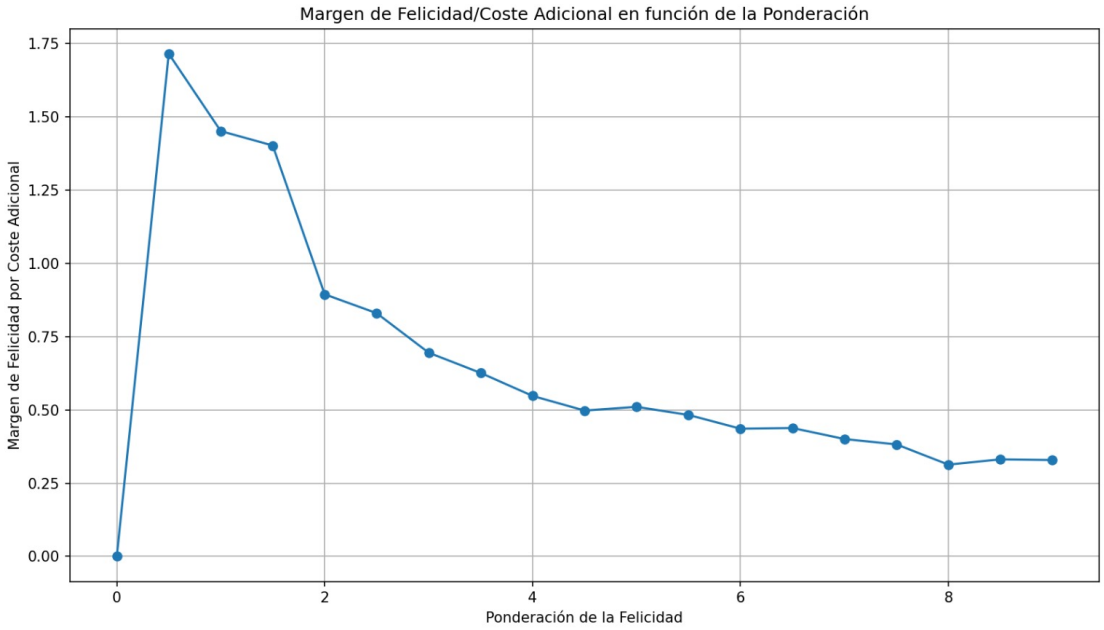


Figure 17: Evolución margen de mejora y ponderación de felicidad

- **Equilibrio Ideal:** En el gráfico, el equilibrio ideal sería una región donde el margen de felicidad por coste adicional se mantenga positivo y relativamente constante. Observando el gráfico obtenido y la tabla de resultados, entre las ponderaciones de 0.5-1.75, el margen se encuentra alrededor de 1.75, LO que nos indica una buena relación entre felicidad y coste adicional. Este rango parece ofrecer un aumento de felicidad significativo sin un incremento excesivo en el coste, lo cual lo hace un punto de equilibrio ideal para maximizar la felicidad de forma eficiente.

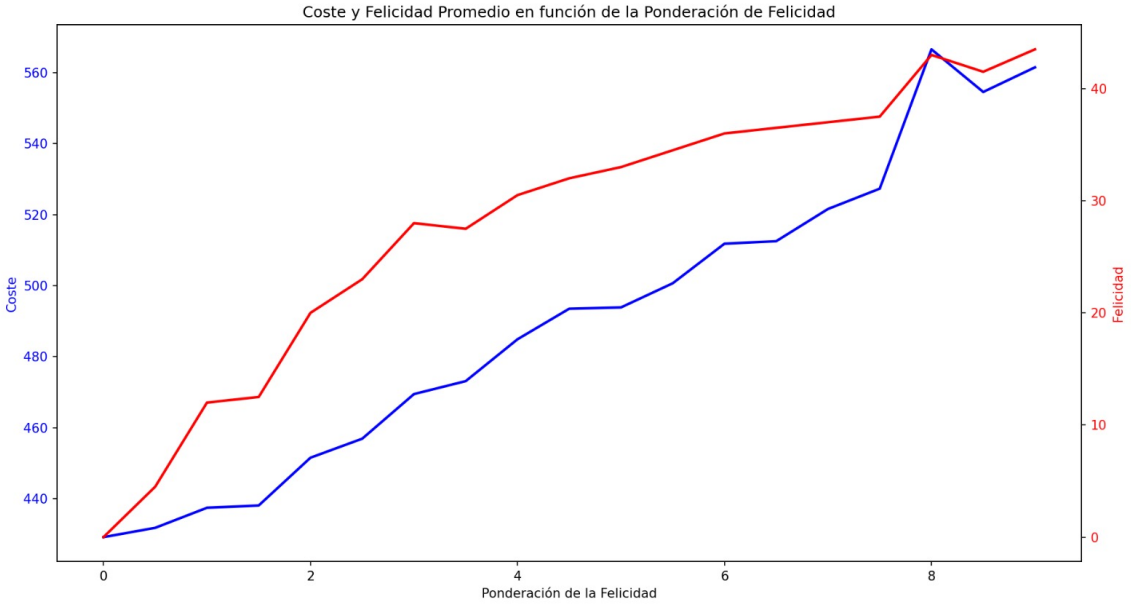


Figure 18: Gráfico evolución coste y felicidad

- **Rango de Oportunidad Moderada:** En el rango de ponderación entre a partir de 2 y antes

de llegar a 4, el margen de felicidad por coste adicional muestra una tendencia a la baja, lo que nos indica que el margen no es mejor que el del equilibrio ideal, manteniéndose en valores de margen de 0.5-0.8, pero observando este segundo gráfico, vemos cómo la felicidad crece mucho más rápida que los costes de la empresa. Esto indica una oportunidad moderada, ya que se podría aprovechar la felicidad generada, aunque el coste empieza a mostrar señales de no mejorar tanto como antes la distribución de la solución inicial, aún así, el coste no es tan elevado todavía. Si buscasemos optimizar prioritáriamente la felicidad pudiendonos permitir elevar costes, este intervalo sería bueno, ya que da valores de felicidad muy altos (valores que van de los 18 a los 32), y el coste se mantiene por la franja de los 500.

- **Desperdicio de Coste en Ponderaciones Altas:** En las ponderaciones más altas, especialmente a partir de 4, el margen nunca alcanza los valores óptimos del equilibrio ideal. Esto nos sugiere que a medida que se incrementa la ponderación de felicidad, el coste adicional ya no es tan eficiente, resultando en un "desperdicio de coste". Aunque el margen no es completamente negativo, los beneficios en felicidad por el coste añadido son mucho menores que en el rango de equilibrio ideal. Podemos ver como el valor del coste sube hasta más de los 520, casi 100 más que inicialmente. Eso sí, los valores de felicidad son altísimos.

6.7.2 Conclusiones

En función a lo que se ha podido observar, se puede llegar a estimar que a medida que la ponderación de felicidad aumente pasará una cosa; la cantidad de felicidad aumentará al igual que también lo hacen los costes. Entonces dependiendo de los objetivos de la empresa se escogería un tipo de ponderación u otra. Si la empresa quiere encontrar priorizar la minimización de costes, se usaría una ponderación de 0.5-1.75, debido a que es la que tiene mejor margen de mejora. Si el objetivo es maximizar la felicidad pudiendo desperdiciar un poco de coste se podría aumentar un poco esta poneración, debido a que el coste no subirá tanto en comparación con la felicidad, que esta como hemos dicho antes crece más rápido. Y si ya no se tiene en cuenta los costes, opatríamos por incrementar la ponderación notablemente para que la felicidad crezca mucho más, pese a que los costes también se elevarían.

6.8 Experimento 8:

- $> 0.25/kg$: Dado que almacenar paquetes en el almacén mientras esperamos a que la empresa de transporte los recoja no resulta rentable, el algoritmo tenderá a dar preferencia a las ofertas que entregan la paquetería en el menor tiempo posible. Esto permite reducir costes de almacenamiento que pueden llegar a ser superiores a los de una oferta inicialmente más cara. Al priorizar las entregas rápidas, se incrementa el índice de felicidad de los clientes, ya que recibir sus compras antes es un incentivo para volver a comprar en el mismo lugar. No obstante, el impacto de esta estrategia dependerá del incremento de coste en cada caso, lo cual puede influir notablemente en el resultado final de la solución.

- $< 0.25/kg$**:** En este caso, resulta más rentable elegir ofertas con plazos de entrega mayores, ya que mantener los paquetes en almacenamiento durante más tiempo es menos costoso que contratar transportes más rápidos y caros. El algoritmo tenderá a ignorar los días de entrega de las ofertas, siempre que se cumpla la restricción de entregar cada paquete a tiempo, y se centrará principalmente en minimizar los precios de las ofertas. Esta estrategia permitiría ahorrar costes al almacenar la paquetería y seleccionar entregas menos inmediatas, aunque los clientes recibirían sus paquetes más cerca del tiempo estimado y la empresa podría renunciar a un posible aumento en la felicidad del cliente. Como en el caso anterior, el impacto dependerá de la variación en los costes, lo cual influirá en el resultado final de la solución.

En resumen, un precio de almacenamiento más alto inclinaría las soluciones hacia entregas más rápidas, lo que reduciría el tiempo de permanencia de los paquetes en el almacén y aumentaría el nivel de satisfacción del cliente. Por el contrario, un precio de almacenamiento más bajo permitiría que el algoritmo mantuviera los paquetes en el almacén por más días mientras selecciona las ofertas con mejor precio, lo que resultaría en menores niveles de satisfacción.

7 Comparación de algoritmos

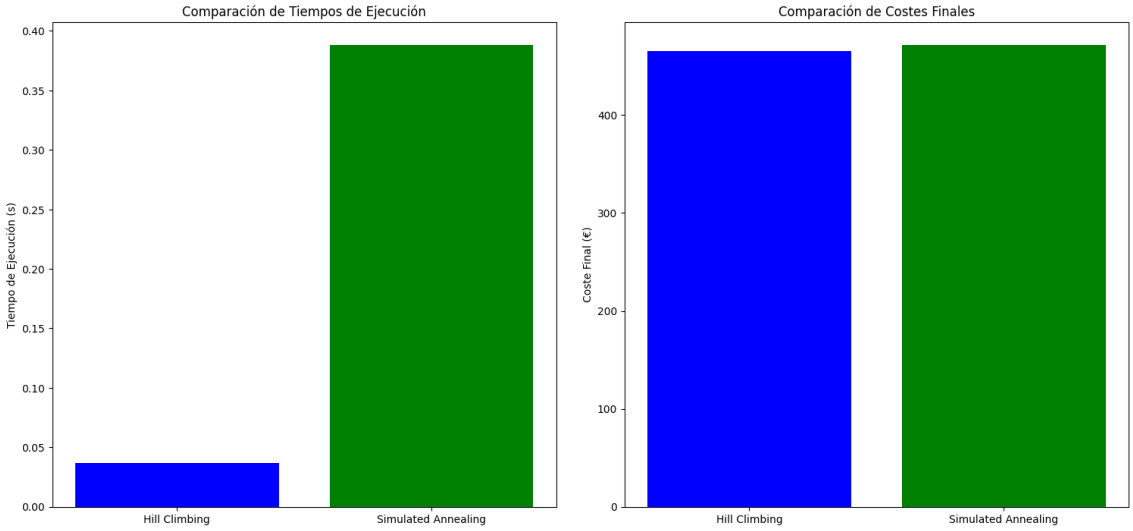


Figure 19: Comparación HC y SA

Hill Climbing mejora ligeramente a Simulated Anneling en términos de coste final, lo que indica que el esfuerzo adicional de búsqueda exploratoria de Simulated Anneling no resulta en una mejor solución en este contexto. Esto sugiere que el espacio de búsqueda y el tipo de problemas con el que nos hemos topado en esta práctica no se beneficia de Simulated Annealing, ya que Hill Climbing ya encuentra soluciones satisfactorias en menos iteraciones.

En cuanto al tiempo de ejecución, Simulated Annealing es considerablemente más lento, siendo aproximadamente de 0.4 segundos frente a los 0.05 segundos de Hill Climbing, lo que representa una diferencia del 500 por cien. Este aumento en el tiempo de ejecución sin una mejora significativa en el coste hace que Hill Climbing sea claramente la opción a elegir en este caso, ya que logra soluciones de calidad similar con una eficiencia mucho mayor en términos de tiempo.