Universidad Politécnica de Catalunya

Inteligencia Artificial

Busqueda Local

Azamon Project

Autores
Pau Celma
Jazmin E. Gell
Ramon Zalabardo

Supervisor
Javier Béjar Alonso

Octubre
QT - 2020/21

Contents

1 Descripción del problema

En esta práctica, se nos plantea una compañía ficticia llamada Azamon que quiere mejorar su método de asignación de paquetes a las ofertas de empresas de paquetería que le hacen los envíos. Se nos presentan una serie de propiedades que tienen los paquetes y las ofertas, que detallaremos más adelante, además de unas restricciones y criterios a seguir cuando generemos y evaluemos una solución con nuestro programa. Además, se nos provee de diversas clases Java

1.1 Elementos del problema

En el problema diferenciamos dos elementos principales: los paquetes y las ofertas. Además, podemos asumir que en parte (si no totalmente) un estado del problema vendrá definido por la asignación de paquetes a diferentes ofertas.

1.1.1 Los paquetes

Un paquete tiene como propiedades su peso y su prioridad. El peso es un número múltiplo de 0,5 kg y no mayor de 10 kg. La prioridad, por otra parte, puede ser de tres tipos: "día siguiente", "2-3 días", "4-5 días". Cada una garantiza que el paquete se entregará en un día, en 3 días y en 5 días como máximo respectivamente. Además, tienen un coste para el usuario (no para la compañía) de 5, 3 y 1.5 euros, respectivamente.

1.1.2 Las ofertas

Una oferta tiene como propiedades su peso máximo, su precio por kilogramo transportado y el número de días en que el paquete llegará a su destino. El peso máximo es un número entre 5 y 50 kg, en intervalos de 5 kg, el precio por kilogramo no tiene ninguna propiedad especial y el número de días estará siempre entre 1 y 5.

1.1.3 Propiedades de una asignación

Una asignación de paquetes a ofertas tiene dos elementos a tener en cuenta en el problema, además de los explicados en los anteriores apartados: el coste de almacenamiento y la felicidad. En los envíos de 3 o 4 días los paquetes se recogen del almacén un día después de ser asignados, mientras que en los envíos de 5 días esto sucede 2 días después. Cada día que un paquete pasa almacenado, Azamon pierde 0.25 euros por kilo. Por otra parte, sabemos que asignar un paquete a un envío que lo vaya a entregar con días de adelanto respecto a la prioridad contratada se traduce en felicidad por parte de los clientes.

1.1.4 Análisis de los elementos

Hay varios elementos en el problema que se relacionan de manera casi intuitiva:

- El peso de los paquetes asignados a una oferta multiplicado por su precio por kilogramo nos dará el coste del envío.
- El peso de un paquete multiplicado por los días que pasa almacenado por 0.25 nos dará el coste de almacenamiento.
- Para calcular el coste total de una asignación deberemos sumar los costes de envío más los costes de transporte.

Respecto a este último punto, uno podría preguntarse si no debería incluirse también el coste de los envíos para los usuarios, que para Azamon representan un beneficio, en el cálculo del coste total de la asignación. Tal y como discutiremos cuando hablemos de los criterios para evaluar la solución, no hemos encontrado necesario incluirlos.

1.2 Restricciones de la solución

Una solución se considera válida si y solo si cumple las siguientes restricciones marcadas por la empresa:

- Un paquete nunca puede llegar más tarde del plazo indicado por su prioridad.
- La suma de los pesos de los paquetes asignados a una oferta no puede superar su peso máximo.
- Todos los paquetes deben estar asignados a una oferta.

1.2.1 Análisis de las restricciones

Una vez añadidas estas restricciones, empezamos a ver más relaciones entre los elementos del problema:

- Deberemos comparar la prioridad de un paquete con los días de su posible oferta asignada para ver si son compatibles.
- Deberemos comparar la suma de los pesos de paquetes asignados a una oferta con el peso máximo de ésta para ver si una solución es válida.

A partir de esto podemos empezar a definir el comportamiento de una posible función que analice si una operación va a llevar o no a una solución válida, a la vez que nos vemos restringidos en el propio tipo de operadores que podremos usar.

1.3 Criterios para evaluar la solución

El enunciado también define dos criterios a seguir para calcular la bondad de una solución:

- La minimización de los costes de transporte y almacenamiento
- La maximización de la felicidad de los clientes

En particular, se nos pide que implementemos dos formas de clasificar asignaciones según su bondad: el coste total de los envíos y una ponderación entre este coste y la felicidad de los clientes.

1.3.1 Análisis de los criterios

Estos criterios nos hacen ver más relaciones entre los elementos del problema:

- Deberemos comparar la prioridad de un paquete con los días de su posible oferta asignada para calcular la felicidad que aporta a la asignación.
- Tal y como hemos indicado antes, para calcular el coste total de una asignación deberemos sumar los costes de envío más los costes de transporte.

Además, habrá que elegir una ponderación entre los dos valores cuando queramos tener en cuenta la felicidad.

1.4 Justificación de la elección de búsqueda local

Una vez analizado el problema, hemos llegado a la conclusión de que debemos modelar el problema como uno de búsqueda local.

La principal razón son los criterios para evaluar la solución que se nos presentan. Se busca optimizar propiedades inherentes al estado del problema, no el camino que se traza para llegar a esa solución. Así pues, las funciones heurísticas de nuestro problema no buscarán aproximar el coste para llegar a un estado, sino que calcularán el valor heurístico de ese estado a partir de las propiedades mencionadas anteriormente.

Además, como veremos más adelante el espacio de búsqueda que forman las soluciones con todos los paquetes asignados es exponencial respecto al número de paquetes. Consideramos que un espacio de búsqueda con todas las posibles asignaciones, válidas o no, sería demasiado grande para pretender encontrar la mejor solución modelando el problema para usar un algoritmo de búsqueda heurística.

2 Implementación del proyecto

2.1 Implementación del estado

En este proyecto, hemos decidido representar el estado del problema como una asignación de paquetes a ofertas. Es decir, que tendremos un estado por cada asignación de paquetes a ofertas. Para ello, hemos usado las siguientes estructuras de datos:

- Un objeto Paquetes `packages`, con la información de los paquetes del problema.
- Un objeto Transporte `transport`, con la información de las ofertas del problema.
- Una *array* de enteros `assignment`, con tantos elementos como paquetes donde el elemento i es igual al índice de la oferta a la que asignamos el paquete i.
- Una *array* de doubles `freespace`, con tantos elementos como ofertas donde el elemento j es igual al peso libre en la oferta j.

Es decir, que si un paquete en `packages` tiene el índice i y está asignado a una oferta que tiene el índice j en `transport`, consultar `assignment[i]` devolverá j. Por otra parte, consultar `freespace[j]` devolverá el peso libre de la oferta, cuyo valor es equivalente al peso máximo de la oferta con el peso de los paquetes asignados a ésta restado.

La lógica detrás de estas estructuras es el ahorro de tiempo. Puesto que la oferta asignada a un paquete es algo que estaremos consultando constantemente, hemos buscado hacer esta consulta más eficiente haciendo que devuelva un entero en lugar de, por ejemplo, un objeto Paquete. Por otra parte, usamos un vector aparte para calcular el espacio libre en cada oferta. Es un dato que estaremos consultando y cambiando constantemente, pero que no está almacenado en ningún lado de la clase Oferta. Así pues, de no almacenarlo tendríamos que recalcularlo cada vez que quisiéramos consultarlo. Como tendríamos que recorrer todo `assignment` para buscar qué paquetes hay asignados en la oferta, este cálculo tendría coste lineal respecto al número de paquetes. Por estas razones, creemos que hemos conseguido nuestro objetivo. Solo hace falta imaginar cómo serían funciones como `swapPackages`, donde se opera usando tanto la asignación como el espacio libre, si no hubiéramos implementado estos dos últimos arrays.

Por último, calculamos el tamaño del espacio de búsqueda. Sabemos que `assignment` tiene n elementos y cada elemento puede tener m valores, donde n es el número de paquetes y m el de ofertas. Por tanto, es fácil ver que el número de posibles combinaciones de valores de `assignment` será de m^n. Ese es el tamaño del espacio de búsqueda que hemos definido.

2.2 Operadores

Hemos implementado dos operadores: `movePackage` y `swapPackages`.

2.2.1 `movePackage(int i, int j)`

Asigna un paquete i en una oferta j. Para poder usarlo hemos de comprobar que la oferta j tenga espacio disponible para que quepa el paquete i. Además de que la prioridad del paquete i no sea superior a la prioridad de la oferta j. Un paquete de prioridad 0 no puede ir a una oferta de prioridad 1. Para ello usamos la función `compatible`.
Los efecto de aplicar nuestro `move` son: actualiza el estado actual guardando el valor j en la posición del paquete i, de tal manera que sepamos que el paquete i va a la ofeta j. Si i antes ya estaba asignado a otra oferta k, entonces actualizamos el espacio disponible de k que ya no tiene paquete i, y actualizamos el espacio libre de la nueva oferta j.

También recalculamos el coste de nuestra solución, para las dos heurísticas. De esta manera cada vez que generamos un estado sucesor no hemos de recorrer toda la solución para calcular el coste.
Move tiene un factor de ramificación de $n * m$, donde n es el número de paquetes y m el de ofertas. Ya que por cada paquete generamos un estado sucesor por cada oferta en el que quepa.

2.2.2 `swapPackages(int i, int j)`

Intercambia las ofertas entre el paquete i y j. Para poder usarlo hemos de comprobar que la oferta del paquete j tenga espacio disponible para que quepa el paquete i y viceversa. Además, de que la prioridad del paquete i no sea superior a la prioridad de la oferta del paquete j, y viceversa. Un paquete de prioridad 0 no puede ir a una oferta de prioridad 1. Para ello usamos la función `compatible`. Al igual que `move`, actualiza los respectivos espacios disponibles de las ofertas y el coste de la solución.
El factor de ramificación es de $((n-1)*n)/2$, donde n es el número de paquetes, ya que por cada paquete (si todos estas asignados) genera un estado sucesor.

Hemos escogido estos operadores porque no generan soluciones no-válidas y nos permiten explorar todo el espacio de soluciones si se usan juntos. El `move` por si solo no nos permite explorar más cuando todos los paquetes estas asignados. El `swap` solo nos permite explorar ofertas que no hayan sido asignadas a paquetes en el estado inicial.

2.3 Estrategias para hallar la solución inicial

En este proyecto hemos decidido usar un algoritmo general para generar las soluciones iniciales, pero hemos definido dos estrategias a partir de la ordenación de los conjuntos de paquetes y ofertas que entramos en el algoritmo. Así pues, hemos definido dos métodos: ordenar los paquetes por prioridad y dejar las ofertas sin ordenar, y ordenar los paquetes por peso y las ofertas por el ratio entre su precio y su peso máximo. La primera no tiene relación con ninguno de los criterios que se describen en el problema. La segunda, por otra parte, va enfocada a dar con una solución inicial buena según el primer criterio.

2.3.1 Algoritmo general

Como algoritmo general hemos usado un backtracking que se queda con la primera asignación viable que encuentra. Hemos escrito un pseudocódigo para explicar su funcionamiento:

```
bool backtrackingAssignment(int[] assignment, double[] freespace, int i)
    Caso base:
    si i >= n_paquetes
        ¡¡Solución encontrada!!

    Paquete p = elemento i de packages

    Caso inductivo:
    para cada oferta j
        si el paquete i es compatible con la oferta j
            assignment[i] = j
            freespace[j] -= p.peso
            si encontramos solución en backtrackingAssignment(assignment, freespace, i+1)
```

```
            ¡¡Solución encontrada!!
        en caso contrario
            assignment[i] = nada
            freespace[j] += p.peso
```

Analizamos la complejidad del algoritmo:
Sea n el número de paquetes a asignar, y m el número de ofertas, podemos expresar $T(n)$ como $T(n) = mT(n-1) + O(1)$. Por tanto, según el teorema maestro I:

$$T(n) = \Theta(m^n)$$

Es lógico que el algoritmo tenga esta complejidad, puesto que es el tamaño del espacio de búsqueda, y los algoritmos de tipo *backtracking* son justamente de búsqueda exhaustiva.

2.3.2 Ordenación por prioridad

Hemos aplicado el algoritmo general sobre una lista de paquetes ordenados por su número de prioridad de forma ascendente. Es decir, que los paquetes de prioridad 0, que son más prioritarios, serán los que se traten primero. Seguidos de los de prioridad 1 y 2. Como es un criterio de ordenación no relacionado con ninguna de las heurísticas que usamos, cuya descripción detallaremos en el siguiente apartado, no se prevé afectar a la bondad de la solución o al tiempo de ejecución de ninguno de los dos algoritmos.

Sin embargo, hemos decidido esta ordenación en lugar de una aleatoria (o ninguna ordenación en absoluto) porque creemos que obligar al algoritmo de backtracking a que trate primero a los paquetes más prioritarios y que, por tanto, tienen un menor número de ofertas con los que son compatibles ayudará a que se tenga que deshacer camino en menos ocasiones dentro del algoritmo.

2.3.3 Ordenación por peso y ratio

Hemos aplicado el algoritmo general sobre una lista de paquetes ordenados por su peso de forma decreciente, de manera que el paquete con mayor peso estará en la primera posición, y una lista de ofertas ordenadas por el ratio entre su precio y el peso máximo de forma creciente.
Esta ordenación nos ayuda a conseguir un coste mas reducido en la solución inicial, ya que los paquetes con mayor peso se asignaran antes a la oferta menos costosa en términos de precio por kilogramos.

2.4 Funciones heurísticas

Factores que intervienen en el problema:

- Kilogramos máximos que puede transportar cada oferta
- Precio por kilogramo transportado según cada oferta
- Número de días en los que los paquetes llegarán a su destino
- Coste de envió según la prioridad del paquete
- Coste almacenaje de kilogramos/día

– Felicidad del cliente

Para la primera heurística no hemos tenido en consideración la felicidad. Tampoco se ha tenido en cuenta en ninguna de las dos heurísticas restar el precio de la prioridad del paquete, ya que este no varía por tanto no aporta información valiosa para poder minimizar.
El enunciado estipula el coste de almacenaje como 0.25 por cada kilogramo/día que el paquete pase en el almacén. También que los paquetes son recogidos dos días antes del envío. Por tanto si la oferta sale en 2 días, el paquete pasará 0 en el almacén.
Para la segunda heurística hemos tenido que investigar como tener en cuenta la felicidad de tal manera que influya en el coste de almacenamiento sin restarle demasiada importancia.

2.4.1 Primer criterio

Hemos planteado un problema de minimización de costes de transporte y almacenamiento.
El coste de un estado es la suma del coste de transporte de cada paquete en base a la oferta a la que está asignado, más el coste de almacenaje del paquete. Más detalladamente para cada paquete es: el peso del paquete por el precio del kilogramo de la oferta la que está asignado + coste de almacenaje. El sumatorio del coste de todos los paquetes es el valor del heurístico de nuestro estado. Nuestra función nos garantiza que si enviar un paquete en otra oferta es más caro el valor de la heurística será mayor al estado actual y el Hill Climbing lo descartará. También refleja que si un paquete se envía antes en otra oferta con el mismo precio, esta valdrá menos porque reducimos el coste de almacenaje. Además antes de cada llamada a `move` y `swap`, garantizamos que los operadores se puean aplicar. Es decir, no excerder el peso máximo de una oferta y respetar las prioridades de los paquetes.

2.4.2 Segundo criterio

La felicidad es el sumatorio del número de días de antelación con los que los paquetes llegan a los clientes.
Para saber los dias de antelación con los que llegan los paquetes restamos el número de dias en los que llega el paquete, según la oferta en la que está, al número de dias mínimos requeridos por la prioridad del paquete.
Para que la felicidad tenga un efecto mayor en el cálculo de la heurística, multiplicamos el valor obtenido por una ponderación. Para calcular el valor del heuristico en nuestro estado restamos el valor obtenido de la felicidad al heuristico que resulta del primer criterio. De esta forma el coste heuristico será mínimo con una felicidad alta y un coste de transporte y almacenaje mínimo.

3 Experimentación

3.1 Estudio de los conjuntos de operadores en Hill Climbing

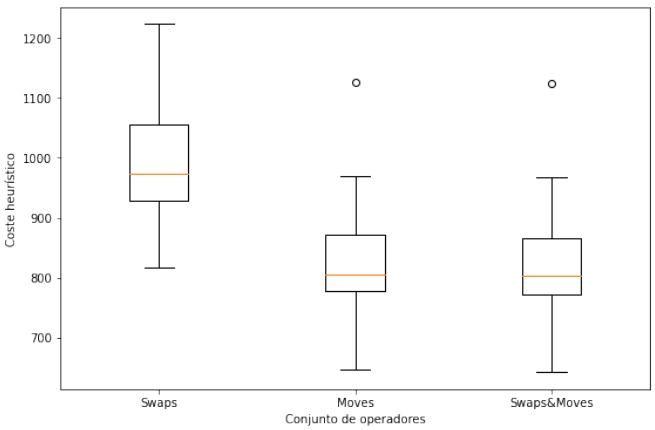
Observación	El conjunto de operadores que usemos al generar sucesores puede influir en la solución final que obtenga el Hill Climbing.
Planteamiento	Usamos 3 conjuntos de operadores: solo *swaps*, solo *moves* y ambos. Observamos el coste y los nodos expandidos del estado final encontrado.
Hipótesis	Usar más operadores nos permitirá llegar a una solución mejor.
Método	• Generaremos 10 semillas aleatorias • Ejecutaremos 3 experimentos para cada semilla, uno por conjunto de operadores • Experimentaremos con un problema de 100 paquetes y proporción de peso 1.2 • Compararemos los resultados obtenidos y determinaremos el conjunto de operadores a usar en próximos experimentos

Una vez hechos los experimentos, hemos recogido los datos del coste y los nodos expandidos por cada solución. Como son relativamente pocos, los hemos reflejado en la siguiente tabla.

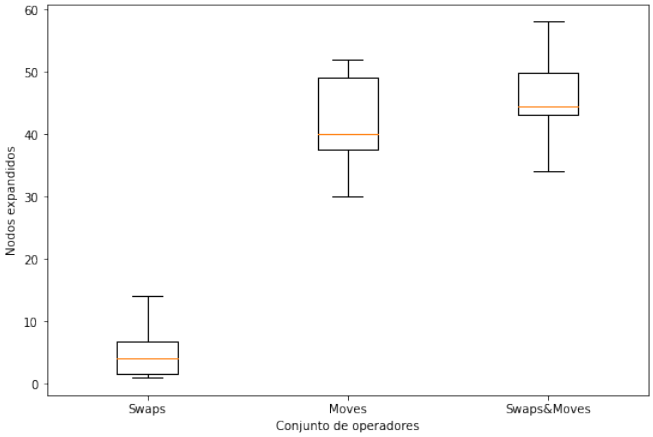
	Coste heurístico			Nodos expandidos		
Semilla	Swaps	Moves	Ambos	Swaps	Moves	Ambos
1234	817.914	647.415	643.225	4	40	46
100	948.635	795.125	792.645	4	39	46
2000	1222.565	1125.755	1123.755	6	30	34
934587	965.975	814.615	812.215	1	36	41
172634	1097.02	968.54	967.6	7	40	43
2345	982.41	804.79	803.75	1	40	43
9876	868.64	689.85	689.13	10	52	53
7583	1030.39	807.31	801.79	3	37	43
2	1064.27	890.16	884.91	1	52	58
42	921.81	772.59	764.88	14	52	51

Figure 1: Datos recogidos en el experimento 1

Dibujamos *boxplots* para poder comparar visualmente los datos de los diferentes conjuntos. Podemos verlos en la figura 2. En lo que respecta a coste heurístico, vemos que operar solo con *swaps* da resultados mucho peores respecto a los otros dos conjuntos de operadores. También podemos observar que visualmente no hay una diferencia clara entre operar solo con *moves* y usar ambos operadores. Respecto al número de nodos expandidos, operando solo con *swaps* expandimos significativamente menos nodos que en los otros dos casos. Además, aquí apreciamos más fácilmente la diferencia entre operar con *moves* y usar ambos operadores, puesto que con *moves* tendemos a expandir menos nodos.



(a) Comparación de los conjuntos de operadores respecto al coste de sus soluciones



(b) Comparación de los conjuntos de operadores respecto a los nodos expandidos en sus soluciones

Figure 2

Tal y como hemos indicado antes, no hay una diferencia apreciable entre el coste heurístico de usar *moves* o ambos operadores. Así pues, hacemos una tabla con la media y la desviación estándar de cada distribución.

	Swaps	Moves	Ambos
Coste	991.963(111.349)	831.615(130.052)	828.39(130.53)
Pasos	5.1(4.061)	41.8(7.25)	45.8(6.431)

Figure 3: Media y desviación estándar de los datos recogidos en el experimento 1

Podemos observar que, con una desviación estándar muy parecida, la media del coste heurístico es menor usando ambos operadores. Además, si nos fijamos en la tabla de la figura 2 podemos ver como en todos los casos el coste es menor al usar ambos operadores.

Así pues, vemos que usar ambos operadores es lo que nos da una solución mejor en cuanto a coste heurístico. Respecto al coste temporal de la búsqueda, el número de nodos expandidos es mayor operando con ambos, y por tanto se deduce que tardará más en acabar la búsqueda. Sin embargo, vemos una relación clara entre número de nodos expandidos y bondad de la solución, así que consideramos que tiene sentido quedarnos con el conjunto de operadores que tardará más si vamos a poder alcanzar una solución mejor.

Concluimos este experimento diciendo que ha resultado como esperábamos. Cuantos más operadores podamos usar, más nodos tendremos en el espacio de búsqueda y más viable será encontrar una solución mejor. Sin embargo, un aumento en el número de operadores implica aumentar el factor de ramificación y expandir más nodos hasta un mínimo local. Así pues, era lógico pensar que usar ambos operadores nos iba a dar la mejor solución pero con el coste temporal más alto.

3.2 Estudio de las estrategias para generar la solución inicial en Hill Climbing

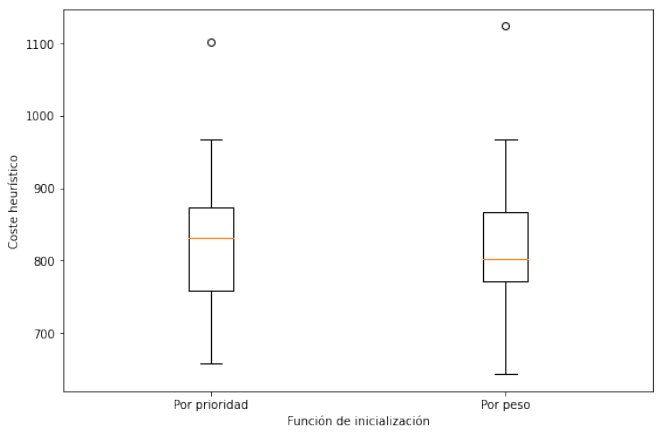
Observación	La estrategia de inicialización que usemos al generar la solución inicial puede influir en el coste temporal del Hill Climbing.
Planteamiento	Usamos 2 estrategias de generación: ordenado por prioridad, ordenado por peso. Observamos el coste y los nodos expandidos del estado final encontrado.
Hipótesis	Hay estrategias de generación que permiten llegar a una solución con menos coste temporal.
Método	• Generaremos 10 semillas aleatorias • Ejecutaremos 2 experimentos para cada semilla, uno por función de inicialización • Experimentaremos con un problema de 100 paquetes y proporción de peso 1.2 • Compararemos los resultados obtenidos y determinaremos la función de inicialización a usar en próximos experimentos

Una vez hechos los experimentos, hemos recogido los datos del coste y los nodos expandidos por cada solución. Como son relativamente pocos, los hemos reflejado en la siguiente tabla.

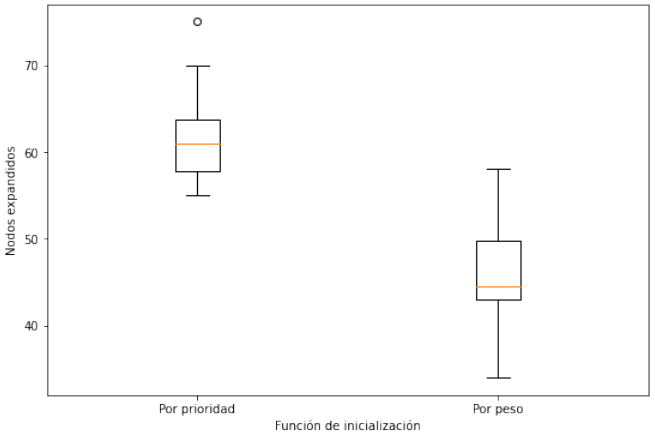
	Coste heurístico		Nodos expandidos	
Semilla	Por prioridad	Por peso	Por prioridad	Por peso
1234	658.355	643.225	60	46
100	753.845	792.645	63	46
2000	1102.36	1123.755	55	34
934587	842.845	812.215	56	41
172634	967.27	967.6	57	43
2345	819.735	8803.75	64	43
9876	702.925	689.13	75	53
7583	850.11	801.79	60	43
2	881.635	884.91	70	58
42	773.395	764.88	62	51

Figure 4: Datos recogidos en el experimento 2

Dibujamos *boxplots* para poder comparar visualmente los datos de las diferentes estrategias. Podemos verlos en la figura 5. En lo que respecta a coste heurístico, vemos que si bien ordenar por prioridad devuelve soluciones ligeramente peores que ordenar por peso, la diferencia no es muy significativa. Respecto al número de nodos expandidos, ordenando por prioridad expandimos más nodos que ordenando por peso. Es decir, que llegamos a la solución en menos tiempo.



(a) Comparación de las funciones de inicialización respecto al coste de sus soluciones



(b) Comparación de las funciones de inicialización respecto a los nodos expandidos en sus soluciones

Figure 5

Para ver numéricamente cómo afecta la elección de estrategia al coste y a los nodos expandidos, calculamos las medias y desviaciones estándar de los datos. Vemos que, efectivamente, hay una ligera mejora en el coste heurístico. Pero donde notamos la elección de estrategia es en el número de nodos expandidos, puesto que este es mucho más bajo cuando ordenamos por peso.

	Por prioridad	Por peso
Coste	835.248(122.761)	828.39(130.53)
Pasos	62.2(5.963)	45.8(6.431)

Figure 6: Media y desviación estándar de los datos recogidos en el experimento 2

Así pues, tanto en coste heurístico como temporal ordenar por peso nos lleva a soluciones mejores que ordenar por prioridad. Por tanto, salir de mejores soluciones iniciales no solo nos lleva a mejores soluciones finales, sino que llegamos en menos tiempo.

Una vez más, esto entra dentro de lo que esperábamos. Partir de una solución generada teniendo en cuenta el criterio heurístico (y que por tanto será mejor) no solo nos llevará más rápido a un mínimo local, sino que este mínimo local podrá ser mejor que si ordenáramos con un criterio no relacionado con la heurística.

3.3 Estudio de los parámetros utilizados en el Simulated Annealing

Variación de las variables K y Lambda

Observación	El valor que tomen los parámetros K y λ influirá en el estado final que obtenga el Simulated Annealing
Planteamiento	Elegimos distintas combinaciones de valores para K y λ, y observamos el coste del estado final encontrado
Hipótesis	Para valores mas altos de λ y más pequeños de K obtendremos peores resultados
Método	• Generaremos 6 semillas aleatorias • Ejecutaremos 12 experimentos por cada semilla para toda combinación posible de K = 1, 5, 25, 125 y λ = 1, 0.1, 0.01 • Haremos las medias de los resultados obtenidos en las 6 ejecuciones de cada combinación • Compararemos los resultados obtenidos

Con el objetivo de analizar que parámetros generaban una mejor solución para el Simulated Annealing, hemos diseñado un experimento que consiste en calcular la media del coste final obtenido en varias ejecuciones, para distintos valores de K y λ. Como podemos observar en la gráfica de la figura XX, la heurística del estado final es relativamente mas grande para todos los casos en los que λ toma por valor 1.0 sin importar el valor de K. A pesar de encontrar un valor mínimo cuando las variables toman por valores $\lambda = 0.01$ y K = 25, como muestra la gráfica la diferencia de los costes obtenidos para las ejecuciones con $\lambda = 0.1$, y $\lambda = 0.01$, es mínima.

Coste heurístico en función de K y λ				
	K = 1	K = 5	K = 25	K = 125
$\lambda = 1.00$	676.66	675.19	686.61	679.73
$\lambda = 0.10$	645.99	646.97	646.96	644.51
$\lambda = 0.01$	642.85	644.81	642.42	642.81

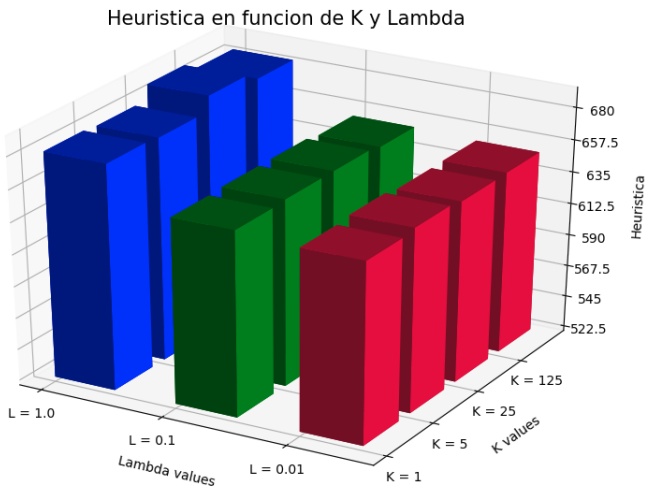


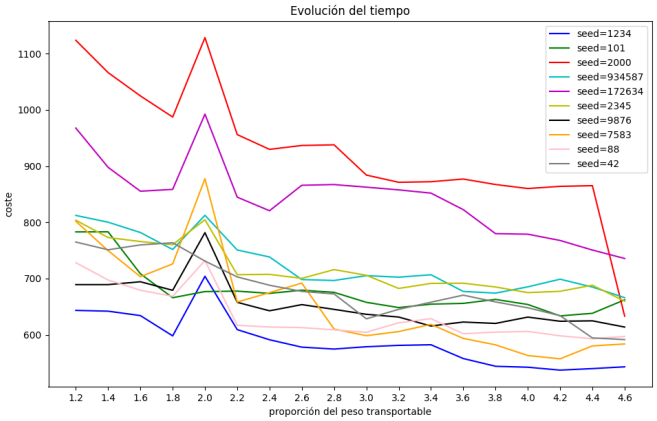
Figure 7: Variación del coste para SA para varios valores de K y λ

Debido a la poca diferencia obtenida para este rango de valores de λ y K, podemos concluir que cualquiera de las combinaciones posibles, dentro de este rango, sería adecuada para obtener una solución óptima para el Simulated Annealing.

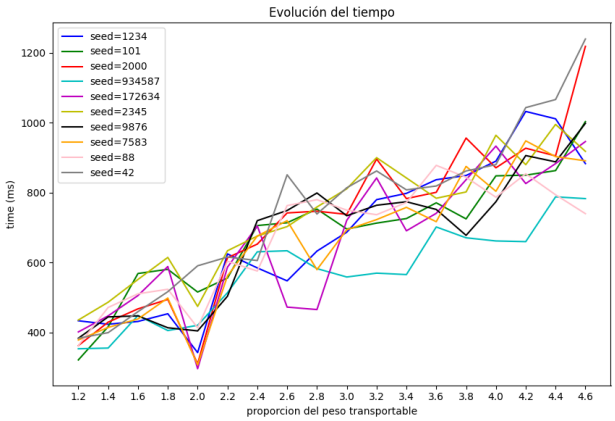
3.4 Variación de la proporción de peso y número de paquetes

Aumento de la proporción del peso transportable	
Observación	Pueden haber proporciones que den mejores resultados de coste y tiempo
Planteamiento	Aumentamos la proporción del peso transportable y observamos el efecto sobre coste y el tiempo
Hipótesis	El variar las proporciones no variara el coste y el tiempo o hay proporciones que sí
Método	• Generamos 10 semillas aleatorias para cada problema • Ejecutamos cada problema incrementando de 0.2 la proporción cada vez hasta notar una tendencia • Mediremos el tiempo de cómputo y el coste final para realizar la comparación • Usaremos el algoritmo de Hill Climbing, 100 paquetes e inicialización eficiente

Una vez realizados los experimentos, hemos recogido los datos de coste, tiempo y nodos expandidos por cada solución. Para cada problema hemos generado un fichero con los resultado. Con un pequeño programa en python hemos leído los ficheros y representado gráficamente el tiempo respecto a p para cada problema.



(a) Evolución del coste en relación a la proporción



(b) Evolución del tiempo en relación a la proporción

Podemos observar una tendencia decreciente del coste respecto la proporción, aunque en $p = 2$ tengamos un incremento. Por lo general decrece, aunque no de manera abrupta. De hecho vemos que aunque doblemos la proporción el coste de la solución decrece poco. Si nos fijamos en lo que ocurre con el tiempo observamos el comportamiento contrario, es decir al aumentar de la proporción el tiempo en hallar la solución se incrementa rápidamente. De hecho si para una $p = 1.4$ teníamos un coste de 700 y un tiempo de 450ms, cuando $p = 4.2$ el coste es 650 y el tiempo 1000ms. Para bajar el coste solo de 50 unidades hemos duplicado el tiempo.
Para asegurarnos de la relación hemos hecho la media de cada problema para cada valor de p y calculado la correlación de Pearson respecto al coste y respecto al tiempo.

$$r_{coste} = -0.91313632 \qquad r_{tiempo} = 0.97453177$$

La correlación de Pearson es un valor acotado entre -1 y 1. Podemos confirmar la relación inversa entre la proporción de p y el coste. También la relación directamente proporcional entre p y el tiempo. Por tanto sí nos afecta el variar p.
Seguramente el crecimiento del tiempo respecto a la proporción de ofertas se debe al factor de ramificación de nuestros operadores. Es decir que al aumentar la cantidad de ofertar, se generar exponencialmente nuevos estados. Por cada nueva oferta podemos realizar un `move` de un paquete a esta y por tanto generar tantos estados nuevos como número de paquetes. Y con `swap` nos ocurre igual.

Aumento del número de paquetes	
Observación	Puede haber un número de paquetes que de mejores resultados de coste y tiempo
Planteamiento	Aumentamos la cantidad de paquetes y observamos el efecto sobre coste y el tiempo
Hipótesis	El variar el número de paquetes no variara el coste y el tiempo o hay cantidades que sí
Método	• Generamos 10 semillas aleatorias para cada problema • Ejecutamos cada problema incrementando de 50 la proporción cada vez hasta notar una tendencia • Mediremos el tiempo de cómputo y el coste final para realizar la comparación • Usaremos el algoritmo de Hill Climbing, proporción 1.2 e inicialización eficiente

Una vez realizados los experimentos, hemos recogido los datos de coste, tiempo y nodos expandidos por cada solución. Para cada problema hemos generado un fichero con los resultado. Con un pequeño programa en python hemos leído los ficheros y representado gráficamente el tiempo respecto a la cantidad de paquetes para cada problema.

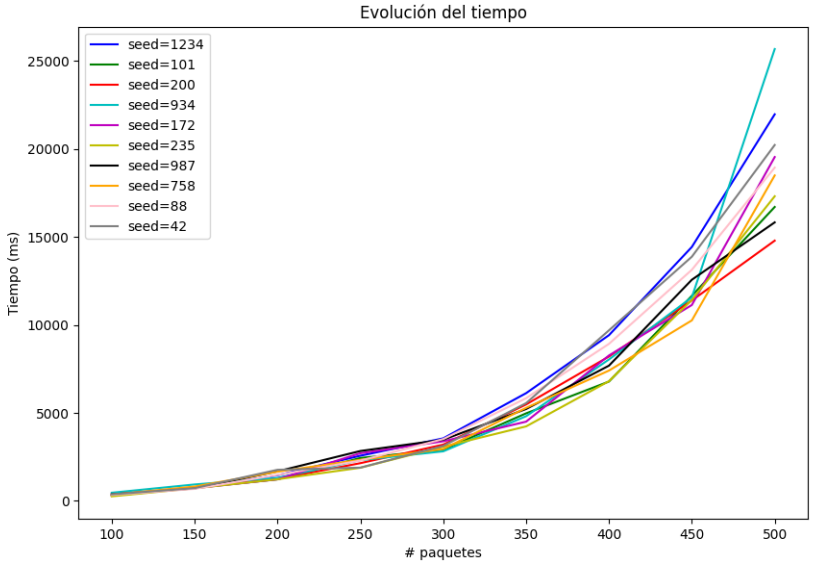


Figure 9: Evolución del tiempo respecto al número de paquetes

Vemos un claro aumento del tiempo respecto al crecimiento de los paquetes. También que al principio con pocos paquetes todos los problemas tardan aproximadamente lo mismo y al crecer este rango se amplia.
Para cerciorarnos hemos calculado la media y la desviación estándar de cada muestra de paquetes.

Tamaño	100	150	200	250	300	350	400	450	500
Media	351.9	753.7	1429.4	2342.6	3202.4	5204.4	8123.6	12144.5	18949.6
Desviación	53.0913	68.99	200.76	299.25	254.93	555.88	951.86	1250.49	3022.32

Figure 10: Media y desviación estándar de los datos recogidos en el experimento 4

Efectivamente, la desviación estándar crece, es decir que el tiempo puede variar más cuanto mayor el número de paquetes. En este caso, el incremento de tiempo también se debe al factor de ramificación de nuestros operadores, hay más posible estados sucesores.
No hemos incluido una gráfica de la relación del coste respecto a los paquetes, ya que claramente están relacionados de manera creciente. El coste de una solución se calcula en base al coste de transportar un número dado de paquetes. Cuantos más paquetes, más costoso es el transporte.
Por tanto aumentar la cantidad de paquetes afecta al coste y al tiempo de nuestra solución.

3.5 Análisis: Comportamiento del coste de transporte y almacenamiento

En los experimento anteriores hemos observado como afecta variar la proporción de ofertas y la cantidad de paquetes al coste y tiempo de nuestra solución. Hemos notado que el incrementar la proporción de la ofertas el tiempo crece casi exponencialmente. De hecho ya habíamos remarcado que el tiempo crecía rápidamente, pero el precio disminuía poco. Gastamos mucho tiempo para poco beneficio. Concluimos pues, que no merece la pena ir aumentando el número de ofertas.

3.6 Efecto de la felicidad de los usuarios en el coste y la ejecución del Hill Climbing

Variación de la ponderación de la felicidad en el cálculo de la heurística

Observación	Variando la ponderación de la felicidad de los usuarios obtendremos distintas soluciones y tiempos de ejecución para el Hill Climbing
Planteamiento	Modificamos el valor por el que se multiplica la felicidad de los usuarios y observamos como afecta en la ejecución y la solución del Hill Climbing
Hipótesis	Todos los valores de la ponderación afectan de igual forma en la ejecución del Hill Climbing y modifican el coste de la solución proporcionalmente
Método	• Generaremos 3 semillas aleatorias • Ejecutaremos 100 experimentos por cada semilla para un rango de valores de la ponderación entre 0 y 10 y haremos las medias. • Observaremos y analizaremos los resultados obtenidos

Para poder analizar bien el efecto de las distintas ponderaciones en la ejecución del algoritmo hemos representado gráficamente la variación del coste de la solución final, el tiempo de ejecución y los nodos expandidos en

función de la ponderación.
En la gráfica que nos muestra la variación del coste podemos ver que este va decrementando linealmente a medida que aumenta la ponderación.
Podemos ver que tanto los nodos expandidos como el tiempo de ejecución se mantienen bastante constantes, en un rango especifico de valores, a medida que crece la ponderación, a excepción de algunos picos que parecen aleatorios. Esto podría indicar que no existe una relación directa entre estos.

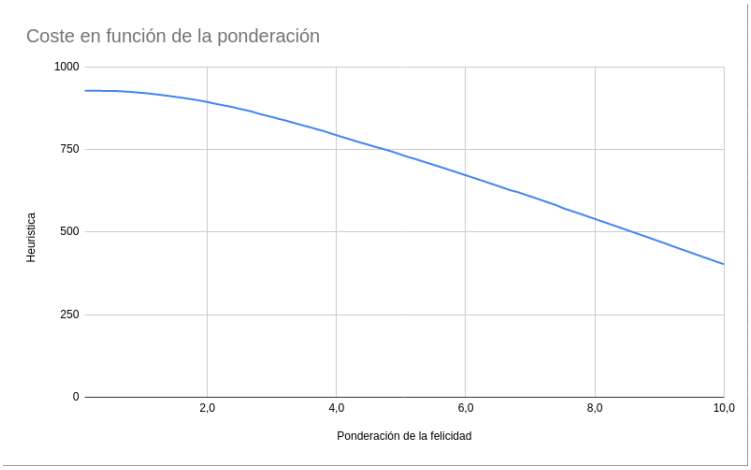


Figure 11: Variación de la heurística en función de la ponderación de la felicidad

Figure 12: Variación de los nodos expandidos en función de la ponderación de la felicidad

El valor de la heurística disminuye de forma linear a medida que aumentamos la ponderación de la felicidad. En cuanto a los nodos expandidos y al tiempo de ejecución, vemos que alcanza los valores mínimos para una ponderación pequeña y se mantiene constante a medida que va aumentando.

Figure 13: Variación del tiempo en función de la ponderación de la felicidad

Debido a esto podemos asumir que elegir una ponderación pequeña de la felicidad implicará una ejecución óptima del algoritmo.

3.7 Efecto de la felicidad de los usuarios en el coste y la ejecución del Simulated Annealing

Variación de la ponderación de la felicidad en el cálculo de la heurística

Observación	Variando la ponderación de la felicidad de los usuarios obtendremos distintas soluciones y tiempos de ejecución para el Simulated Annealing
Planteamiento	Modificamos el valor por el que se multiplica la felicidad de los usuarios y observamos como afecta en la ejecución y la solución del Simulated Annealing
Hipótesis	Todos los valores de la ponderación afectan de igual forma en la ejecución del Simulated Annealing y modifican el coste de la solución proporcionalmente
Método	• Generaremos 3 semillas aleatorias • Ejecutaremos 100 experimentos por cada semilla para un rango de valores de la ponderación entre 0 y 10 y haremos las medias. • Observaremos y analizaremos los resultados obtenidos

Para analizar los resultados obtenidos en los experimentos, hemos procedido de forma similar al experimento 6. Dado que en el Simulated Annealing siempre se expande el mismo número de nodos, a diferencia del Hill Climbing, era innecesario generar la gráfica de los nodos expandidos.
En la gráfica que representa el coste en función de la ponderación de la felicidad, podemos ver que no se diferencia en gran cantidad de la gráfica del Hill Climbing. Ambas decrecen linealmente a medida que se aumenta la ponderación de la felicidad.

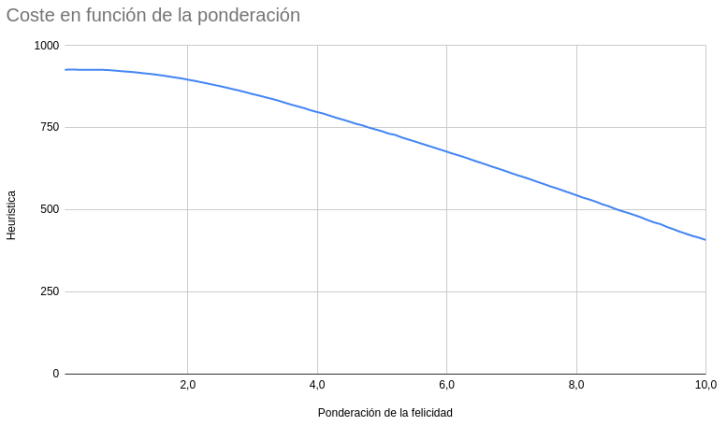


Figure 14: Variación de la heurística en función de la ponderación de la felicidad

Sin embargo en cuanto al tiempo de ejecución, si comparamos las figuras 13 y 15, hay una diferencia bastante notable entre el Hill Climbing y el Simulated Annealing. A pesar de los picos que se dan con una ponderación baja, el tiempo de ejecución del Simulated Annealing se estabiliza sobre los $70ms$ cuando la ponderación toma valores mayores que 3, en cambio en la ejecución del Hill Climbing se estabiliza sobre los $115ms$.

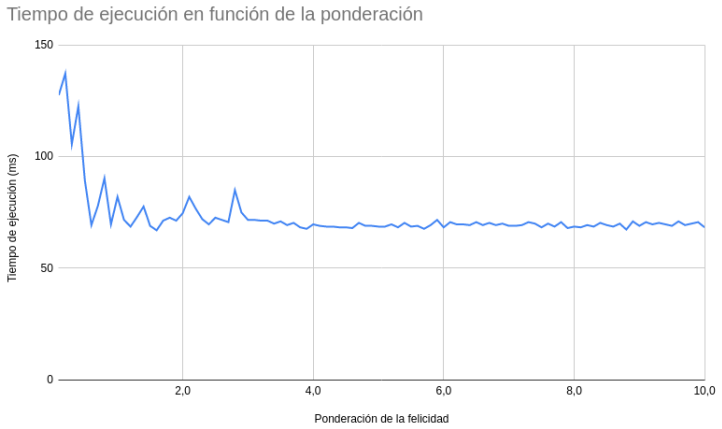


Figure 15: Variación del tiempo de ejecución en función de la ponderación de la felicidad

3.8 Efectos de la variación del precio de almacenamiento en las soluciones

¿Como afectará el cambio del precio de almacenamiento a las posibles soluciones obtenidas por ambos algoritmos?

Teniendo en cuenta que todos los paquetes asignados a envíos de 3, 4 o 5 días permanecerán al menos un día almacenados, podemos deducir que el precio de almacenamiento tendrá una gran importancia en el coste total de las soluciones.

En caso de que incrementáramos el precio de almacenamiento, el coste de la solución final también aumentaría, y en el caso contrario disminuiría.

4 Comparación de los dos algoritmos

Después de haber realizado los experimentos anteriores podemos proceder a comparar los algoritmos Hill Climbing y Simulated Annealing para concluir cual de los dos es más eficiente para el problema tratado en este trabajo.
Para comparar ambos algoritmos nos vamos a centrar en dos puntos de estos, el coste temporal de la búsqueda y la bondad de las soluciones.
Como hemos observado en el experimento 3, después de haber explorado los valores de los parámetros utilizados en el Simulated Annealing, hemos concluido que los valores para K y λ deben ser 25 y $0,01$ respectivamente, para maximizar la eficiencia del algoritmo.

4.1 Coste temporal de la búsqueda

Como hemos observado previamente en los experimentos 6 y 7, el algoritmo Hill Climbing parece tardar más en la ejecución de la búsqueda. En las graficas de las figuras 13 y 15 se puede ver como el algoritmo Hill climbing suele tardar unos $115ms$, en cambio en el algoritmo Simulated Annealing suele tardar unos $70ms$.
Dado a que esos experimentos no se centran en la comparación del tiempo entre ambos y pueden afectar otros factores, hemos decidido hacer un pequeño experimento para comprobar si realmente el algoritmo Simulated Annealing tiene un menor tiempo de ejecución que el Hill Climbing.
Para este experimento hemos ejecutado ambos algoritmos para 10 semillas distintas y con el mismo número de paquetes y proporción del peso transportable.

Tiempo de ejecución para las diferentes seeds (ms)		
Seed	Hill Climbing	Simulated Annealing
1	712	276
2	596	305
3	517	263
4	505	282
5	729	283
6	700	317
7	686	335
8	656	333
9	660	328
10	516	419
Media	627,7	314,1

Como se puede ver en la tabla anterior, el Simulated Annealing tarda epracticamente la mitad que el Hill Climbing en encontrar la solución final para todas las semillas.
La media de los tiempos de ejecución para el Simulated Annealing es 0,5004 veces la media de los tiempos de ejecución para el Hill Climbing.

4.2 Bondad de las soluciones

En cuanto a la bondad de las soluciones, hay que tener en cuenta que el Simulated Annealing recibe un valor que indica cuantas iteraciones debe hacer por lo que el tiempo de ejecución de este variara dependiendo del número de ejecuciones por lo que la comparación anterior no seria muy representativa si estas ejecuciones no obtuvieran

soluciones igual o mas buenas que el Hill Climbing. También hemos obtenido el coste del estado final de las ejecuciones anteriores para poder compararlos.

Tiempo de ejecución para las diferentes seeds (ms)		
Seed	Hill Climbing	Simulated Annealing
1	1076,88	1076,88
2	1013,34	1013,01
3	957,91	957,91
4	1038,66	1036,86
5	859,40	859,40
6	967,92	967,92
7	852,67	852,67
8	1058,15	1058,14
9	1008,55	1007,54
10	1041,58	1038,51
Media	987,51	986,89

Como se puede ver en la tabla anterior, el Simulated Annealing no solo obtiene soluciones con el mismo coste que el Hill Climbing, sinó que algunas de estas son mejores.
Despues de haber comprobado que el Simulated Annealing llega a las mismas soluciones que el Hill climbing, e incluso a mejores, en la mitad de tiempo que el Hill Climbing, podemos concluir que el algoritmo de Simulated Annealing es más eficiente a la hora de encontrar soluciones para este problema.

Appendices

A Proyecto de innovación

A.1 Descripción tema:

Leap motion es un sensor capaz de detectar el movimiento de las manos y los dedos de manera que el usuario pueda interactuar con la pantalla sin tener que tocar nada. Utilizando conocimientos de inteligencia artificial y hardware ha revolucionado la experiencia del usuario respecto a como relacionarse con la tecnología.

A.2 Reparto trabajo

Usando los apartados indicados en el proyecto hicimos un esquema del documento. Buscamos cada uno por su cuenta distintas fuentes de información. Después decidimos que querríamos comentar en cada apartado de manera más detallada y nos repartimos las secciones del documento. Cada cual redacto su parte y revisó la de los demás. Nos dimos una deadline para dar sugerencias. Reeditamos nuestro documento LaTeXy miramos faltas de ortografía.

A.3 Referencias

[1] Crunchbase: Leap Motion,
https://www.crunchbase.com/organization/leap-motion

[2] ultraleap: Leap Motion,
https://www.ultraleap.com/

[3] ultraleap: DataSheets,
https://www.ultraleap.com/datasheets/Leap_Motion_Controller_Datasheet.pdf

[4] Wikipedia: Leap Motion,
https://en.wikipedia.org/wiki/Leap_Motion

[5] Youtube: Introducing the Leap Motion,
https://www.youtube.com/watch?v=_d6KuiuteIA

[6] Gesture Recognition For Smart TV Market to Witness Huge Growth by 2028 — Eyesight Tech, Leap Motion, LG Electronics, Panasonic,
https://www.marketwatch.com/press-release/gesture-recognition-for-smart-tv-market-to-witness-huge-growth-by-2028-eyesight-tech-leap-motion-lg-electronics-panasonic-2020-10-01

[7] Engineering.com: A Major LEAP for Motion Capture Technology,
https://www.engineering.com/DesignerEdge/DesignerEdgeArticles/ArticleID/18201/A-Major-LEAP-for-Motion-Capture-Technology.aspx

[8] Engineering.com: Artificial Intelligence Learns from the Animal Kingdom,
https://www.engineering.com/DesignerEdge/DesignerEdgeArticles/ArticleID/15733/Artificial-Intelligence-Learns-from-the-Animal-Kingdom.aspx

[9] The Verge: Leap Motion, the gesture startup reportedly almost acquired by Apple, sells to UK haptics company, https://www.theverge.com/2019/5/30/18645604/leap-motion-vr-hand-tracking-ultrahaptics-acquisition-rumor

[10] GitHub: Leap Trainer, https://github.com/roboleary/LeapTrainer.js

[11] Rawgithub: Leap Trainer, https://rawgithub.com/roboleary/LeapTrainer.js/master/trainer-ui.html

[12] Leap Motion Developer, https://developer.leapmotion.com/documentation

[13] Researchgate, https://www.researchgate.net

A.4 Dificultades

Sobretodo encontrar información sobre el tipo de IA que han utilizado y los algoritmos implementados.