Descripción de las estructuras de datos y los algoritmos del sistema

Sergio Polo
Sambhav Mayani
Joan Gomez
David Calvo

Asignatura: PROP
Cuatrimestre Otoño 2024-2025

Índice

Antes que nada, mencionar que este documento es acumulativo y sirve como base para cada una de las entregas. Además, está pensado como documentación para todos los miembros del equipo con el fin de entender cada parte del sistema aquí documentada.

1. Introducción

1.1. Introducción al problema

El problema que se intenta resolver consiste en la ordenación circular de una cantidad de productos relacionados entre sí de forma que la relación entre dos productos contiguos sea máxima. Donde no se pueden repetir productos en la ordenación resultante y la relación entre dos productos cualquiera está acotada entre [0,1].

Podemos ver una similitud con los problemas denominados Travelling Salesman Problem, cuyo fin es encontrar un camino, en un grafo con pesos no dirigido, cuya distancia total recorrida sea mínima pero en nuestro caso se exige que sea máxima. Veamos pues la similitud entre ambos tipos de problemas. En primer lugar, la ordenación de productos circular se puede asemejar a un camino a recorrer donde cada producto equivale a un vértice y dos productos contiguos equivale a una arista del camino entre los dos productos, el peso (coste) de una arista equivale a la relación entre dos productos, y finalmente que en la ordenación no aparezcan dos productos repetidos equivale a no repetir vértices en el camino que nos evitará la aparición de ciclos en la ordenación resultante. Vista la semejanza, el problema se puede representar como un grafo completo puesto que se nos garantiza que todas las relaciones entre dos productos cualquiera están definidas excepto a sí mismo.

Se conoce que los problemas de la categoría TSP son intratables. En particular, pertenecen al conjunto de problemas NP-completos y se desconoce, todavía, una solución polinómica a estos problemas. Esto quiere decir, que no existe un algoritmo que resuelva estos problemas en tiempos polinómicos y que por tanto, cuanto mayor sea el tamaño del grafo en cuestión mayor será el tiempo para resolverlo, creciente de forma exponencial. Llegando incluso a tiempos intratables (de miles de años) para problemas relativamente pequeños.

En este sentido, debemos acontentarnos con soluciones subóptimas, que se pueden calcular en tiempo polinómico y que, en una cota aproximada, se aproximan a la solución óptima. Es

objeto de este escrito, analizar las diversas estrategias que incorpora el sistema para resolver el problema en cuestión.

1.2. Equivalencia entre soluciones mínimas y máximas

Un importante punto a comentar es que el problema TSP es a minimizar, es decir que busca el camino de distancia mínima, mientras que el problema a resolver es a maximizar, es decir que busca el camino de distancia máxima. Esto no supone un problema porque, para un vértice cualquiera, en vez de escoger la arista mínima de entre todas sus adyacentes, escogemos aquella que es máxima. Como existe linealidad entre ambos problemas, se puede aplicar la misma técnica de resolución para ambos problemas únicamente cambiando la condición de aceptación. Dicho esto, como el objetivo del sistema es resolver un problema de categoría TSP, mantendremos la afinidad con el resto de documentación del ámbito de la computación reconvirtiendo el problema original a minimizar. Para esto, consideraremos que el peso de una arista consiste en $1 - relación(V_i, V_j)$ para dos vértices cualquiera. del grafo. Pues, cuando dos productos están muy relacionados, su $relación(P_i, P_j)$ es próxima a 1, por tanto, tras la reconversión, el peso (coste) de la arista entre estos será muy pequeño. De esta forma, buscar el camino mínimo con estos pesos es equivalente a buscar el camino máximo.
Únicamente en la capa de presentación mostraremos el peso (relación) entre dos productos revirtiendo la transformación.
Podemos aplicar esta transformación porque existe linealidad entre los dos tipos de expresiones, de forma que incluso para el coste de la solución final podemos aplicar dos transformaciones constantes que permiten pasar de un coste total mínimo a máximo. A continuación, presentamos la relación lineal entre ambas.

El coste de la solución mínima del problema consiste en la suma de pesos de todas las aristas que forman el camino mínimo, esto es:

$$exp1\colon\ (1 - p_1) + (1 - p_2) + \cdots\cdots + (1 - p_n)$$

$$donde \quad p_n = relación(V_i, V_j) \quad para\ dos\ vértices\ contiguos$$

Mientras que el coste que buscamos es:

$$exp2\colon\ p_1 + p_2 + \cdots\cdots + p_n$$

$$donde \quad p_n = relación(V_i, V_j) \quad para\ dos\ vértices\ contiguos$$

Manipulando la $exp1$ podemos llegar a la $exp2$.

$$exp1: (1 - p_1) + (1 - p_2) + \cdots\cdots + (1 - p_n) =$$
$$= 1 - p_1 + 1 - p_2 + \cdots\cdots + 1 - p_n =$$
$$= n + (-p_1 - p_2 - \cdots\cdots - p_n) =$$
$$= n - (p_1 + p_2 + \cdots\cdots + p_n)$$

De esta forma, si a $exp1$ le restamos n y luego multiplicamos por -1 o primero multiplicamos por -1 y luego le sumamos n, entonces obtenemos $exp2$.

Así, podemos obtener el coste final máximo aplicando dos operaciones constantes de coste $O(1)$.

2. Descripción de las Estructura de Datos del sistema

2.1. Representación del grafo

En primer lugar, haremos uso de la palabra peso de una arista en un grafo como el coste entre los dos vértices que une la arista. Recordemos que el coste de una arista, tras la reconversión, se expresa como:

$$(1 - p_n)$$
$$donde \quad p_n = relación(V_i, V_j) \quad para\ dos\ vértices\ contiguos$$

Para poder resolver el problema, debemos determinar cuál será la representación del grafo con pesos no dirigidos. Utilizamos la estructura de datos de un *vector<vector<double>>* (en términos de Java: double[][]) cuyo tamaño de cada vector es el número de productos del sistema para representar las relaciones entre los productos como una matriz NxN de pesos donde para cada producto se define el peso de las aristas con todo el resto y un producto tiene peso máximo consigo mismo. Llamaremos a esta estructura de datos una tabla de similitud.

Existen dos posibles implementaciones para un grafo con pesos no dirigido, escogemos la matriz de adyacencia sobre la lista de adyacencia porque el coste de consulta del peso de una arista es constante mientras que para la lista es lineal $O(n)$. Además, pese a que el coste espacial de la matriz de adyacencia pueda ser $O(n^2)$, debido a que en el problema tratamos con un grafo completo, están definidas todas posibles aristas del grafo, y por tanto, el coste

espacial de la lista de adyacencia que es $O(n + m)$ pertenece a $O(n^2)$ al ser $m = C_2^n = \frac{n \cdot (n-1)}{2} = \frac{n^2 - n}{2}$.

Por esto, consideramos que el uso de la matriz de adyacencia es significativamente mejor que la lista de adyacencia para este problema en concreto.

2.2. Utilización de estructuras eficientes

Debido a que las matrices se crean sobre índices de números naturales, utilizaremos HashMap<idProducto, int> para agilizar la consulta de identificadores de productos a los vértices del grafo. De esta forma, el coste de acceso a un vértice sigue siendo constante puesto que el coste de acceso a un elemento del mapa es $O(1)$. Además, con esta estructura el sistema puede permitir la creación de múltiples tablas de similitudes para los mismos objetos.

2.3. UnionFind

El Union Find es una estructura de datos muy eficiente que almacena una partición dinámica de un conjunto, o dicho de otra manera, los subconjuntos disjuntos de un conjunto.

Como su nombre indica, tiene 2 operaciones principales.

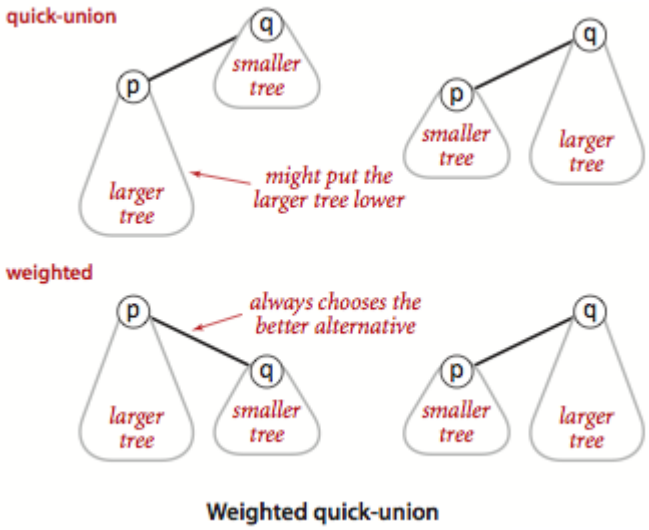
- *Union:* Unir 2 subconjuntos disjuntos.
- *Find:* Buscar a qué subconjunto disjunto pertenece un elemento.

Para la implementación de esta estructura, utilizaremos un árbol que represente a cada nodo disjunto, usaremos un vector de id[] que para cada elemento guarda a su padre inmediato en el árbol que representa al subconjunto en el que está, entonces para saber si 2 nodos pertenecen al mismo subconjunto hemos de comparar el padre de sus respectivos árboles (recorriendo el árbol ascendientemente), y para hacer la unión de dos subconjuntos hemos de "colgar" el árbol de un subconjunto al árbol de otro.

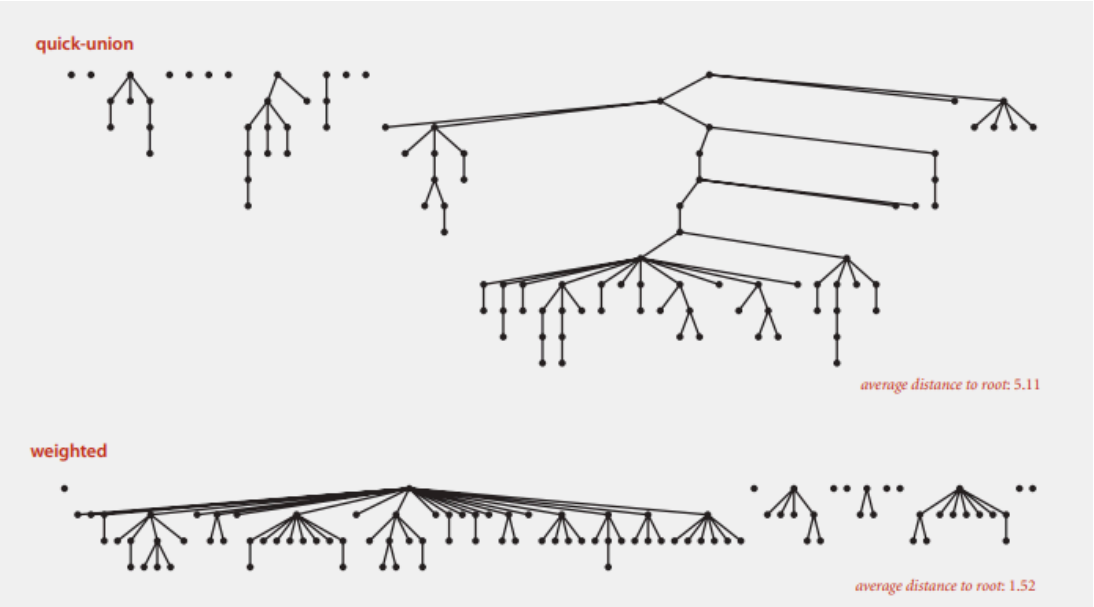
A continuación voy a explicar un poco más en detalle el *Union* y el *Find* y las optimizaciones que hemos implementado para que usar ambas funciones tenga un coste prácticamente constante.

2.3.1. Union Optimizado

Para optimizar al *Union*, con el que básicamente unimos los 2 árboles representantes de cada subconjunto, lo que haremos es asegurarnos de que el árbol más bajo "cuelgue" del más alto, guardándonos el tamaño de cada árbol en un vector para así poder compararlos. Ya que como se ilustra en la imagen de abajo, todos los vértices del árbol que escogemos para que cuelgue del otro ganan un grado extra de profundidad, y por lo tanto, cada vez que consultamos uno de estos vértices para saber el padre de su árbol y identificar así a qué subconjunto disjunto del UnionFind pertenece, tendremos que subir un nivel extra, lo cual es un coste añadido que se va acumulando para cada *Union* que hagamos.



Esquema representativo del *Union* Optimizado, se puede observar la mejora de la parte de abajo respecto a la de arriba, ejemplificando así el porqué es beneficioso colgar el árbol pequeño al árbol grande
Fuente: https://algs4.cs.princeton.edu/15uf/



Otra imagen donde se puede ver como los árboles de abajo quedan mucho más bajos aplicando la optimización.

Fuente: https://algs4.cs.princeton.edu/lectures/keynote/15UnionFind-2x2.pdf

Aquí está el pseudocódigo:

Union (vértice x, vértice y)

padreX ← find(x)
padreY ← find(y)
si padreX es distinto de padreY
  si el árbol del subconjunto de X es más grande que el de Y
  cuelgo el árbol del subconjunto de Y al de X
  si no
  cuelgo el de X a Y

Figura 1: Union Optimizado

2.3.2. Find Optimizado

Al igual que el *Union*, lo que buscaremos es acortar la altura de los árboles que representan a cada subconjunto disjunto, para eso, cada vez que llamemos a *Find* para un vértice y busquemos el padre del árbol que representa a su conjunto, comprimiremos el camino que separa a ambos nodos, como ilustra la imagen de abajo.

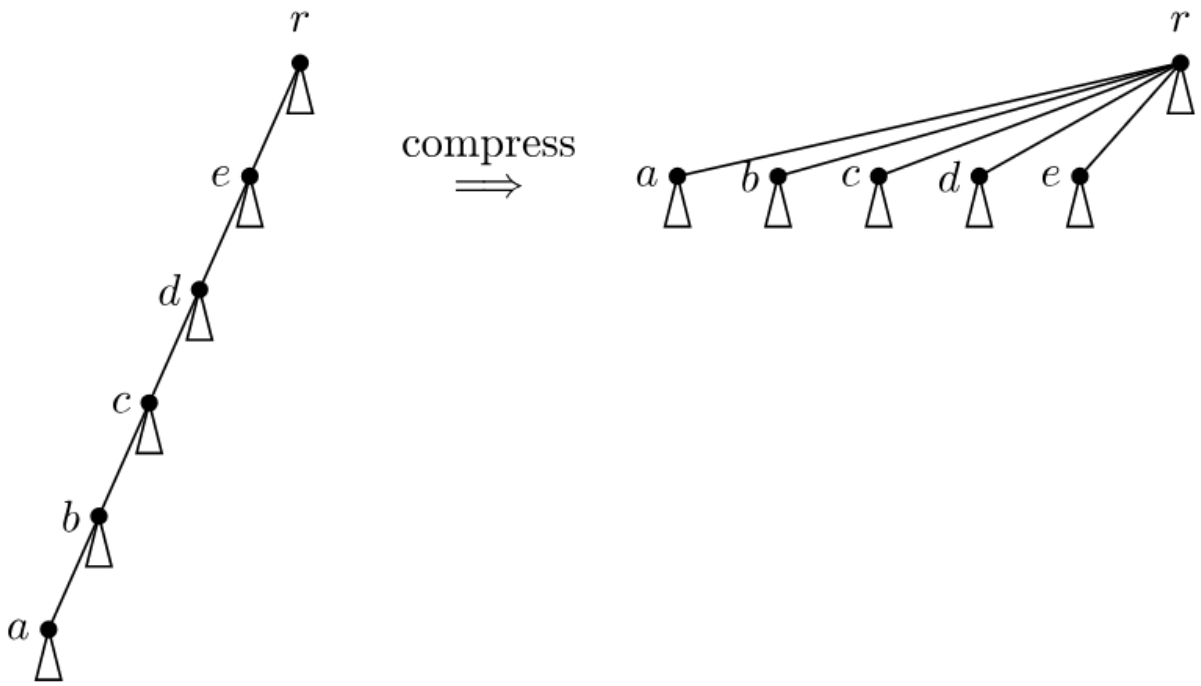


Imagen en la que podemos observar los efectos de la compresión de camino.

Fuente: https://kubokovac.eu/gnarley-trees/figs/union-find/compress.svg

Este es el pseudocódigo, se puede ver cómo se hace una compresión de camino recursiva del vértice de entrada al vértice padre de todo el árbol del subconjunto todos los nodos en el camino también los vamos comprimiendo.

Find (vértice x), id[|V|] es un parámetro de la estructura de datos de UnionFind
si padreX es distinto de padreY si x es distinto de id[x] (su padre inmediato en el árbol) id[x] ← find(id[x]) x ← id[x] retorno x

Figura 2: Find Optimizado

2.4. Resumen

- **double[][]**: Matriz de relaciones

  Coste Espacial: O(n^2)

  Coste Temporal de acceso: O(1)

- **HashMap<idProducto, int>**: Guarda para cada producto la posición en la matriz de relaciones.

  Coste Espacial: O(n)

  Coste Temporal de acceso: O(1)

- **UnionFind**: Usado en Kruskal.

  Coste Espacial: O(n)

  Coste Temporal de acceso *Find* y *Union*: O(1) se puede considerar constante, la función de coste crece extremadamente lento.

n = número de productos

3. Descripción de los Algoritmos del sistema

Debido a la intratabilidad del problema para cantidades medianas y elevadas de vértices, el uso de algoritmos de fuerza bruta como estrategia para obtener la solución óptima del problema es inviable puesto que no acabarían en un tiempo razonable. Esto es porque a pesar de que el algoritmo garantice encontrar la solución óptima, itera sobre todo el conjunto de posibles soluciones hasta dar con esta, y este puede ser muy grande. Además, aunque se apliquen podas al algoritmo, el coste temporal de este sigue siendo exponencial. Por estos motivos, hemos decidido aplicar algoritmos de menor coste temporal que nos garanticen soluciones subóptimas y resuelven en tiempo polinómico.

A partir de ahora utilizaremos coste y peso entre dos vértices indistintamente, y definimos que el coste mínimo es la suma de pesos de las aristas que forman el camino mínimo. Se dejará claro la diferencia entre el coste del camino y el coste o complejidad del algoritmo.

3.1. Nearest Neighbor

El Nearest Neighbor (NN) es un algoritmo voraz cuya idea fundamental consiste en, para un vértice cualquiera, coger el vecino más cercano. Este algoritmo no exige que se cumpla la desigualdad triangular en el grafo.

En detalle y por pasos:

1. Se elige un vértice inicial.
2. Para todos los vértices adyacentes se obtiene la arista cuyo peso sea el mínimo.
3. El siguiente vértice del camino será el vértice adyacente de la arista seleccionada.
4. Se itera sobre el nuevo vértice seleccionado hasta haber visitado todos los vértices.

El algoritmo NN garantiza soluciones válidas pero no óptimas, además la mayoría de sus implementaciones son de coste $O(n \cdot (n-1)) \in O(n^2)$ puesto que para cada vértice del grafo recorre todo sus vértices adyacentes, y en caso peor recorrerá todos los vértices del grafo excepto a sí mismo.
El Nearest Neighbor se conoce como el algoritmo voraz más eficiente para resolver problemas de tipo TSP, pese a que no garantice soluciones óptimas y tenga una pérdida de coste del camino de hasta el 25 hasta el 40% para problemas de tamaños elevados [1] [2], el uso del NN es una buena aproximación para resolver problemas de tamaño pequeño o mediano, debido a que su eficiencia temporal y la bondad de su solución son muy similares a otros algoritmos más costosos.
Nótese que el NN no es determinista puesto que el camino que obtiene puede no ser mínimo y suele cambiar dependiendo de cual sea el vértice inicial.

El sistema **incorpora una versión optimizada** del algoritmo que reduce el coste $O(n^2)$ a términos lineales. Esto es posible ya que, si por cada vértice del grafo, en vez de recorrer todos los vértices adyacentes excepto él mismo recorremos solo los vértices adyacentes que todavía no aparecen en la solución (es decir, excluimos los que ya han sido visitados pues no pueden volver a aparecer), el coste final se convierte en $O(n)$. Véase el pseudocódigo del algoritmo de la Figura 1.

```
Nearest Neighbor Opt. (grafo, verticeInicial)
verticesAVisitar ←lista[grafo.tamaño] excepto el vértice inicial
sol ← vector<naturales>[grafo.tamaño]
indiceSol = 0; sol[indiceSol] ←verticeInicial; indiceSol = indiceSol + 1;
mientras no verticesAVisitar.vacia():
        verticeNN = verticesAVisitar[0]
        min = Ady(sol[indiceSol], verticesAVisitar[0]).peso
        i = 1
        mientras i < verticesAVisitar.tamaño :
            c = Ady(sol[indiceSol], verticesAVisitar[i]).peso
            si c < min entonces:
                min = c
                verticeNN = verticesAVisitar[i]
            fin si
            i = i + 1
        fin mientras
        sol[indiceSol] = verticeNN; indiceSol = indiceSol + 1;
        verticesAVisitar.eliminar(verticeNN)
fin mientras
devuelve sol
```

Figura 3: Algoritmo Nearest Neighbor Optimizado

Observamos que si implementamos el algoritmo con iteradores, la operación de eliminar sobre la lista de vértices a visitar se vuelve de coste constante. Debido a que ya no se visitan los mismos vértices cada vez, sino que consultamos 1 vértice menos cada iteración de forma sucesiva, obtenemos un nuevo coste del algoritmo que es lineal de la forma:

$$C = O(costeInicializacionLista + costeRecorrerLista)$$

$$O(n + ((n-1) + (n-2) + \cdots\cdots + (n - (n-1))))$$

$$Eq.\ O(n + ((n-1) + (n-2) + \cdots\cdots + 1))$$

Donde definimos la siguiente sucesión:

$$a_0 = n - 1$$

$$a_{m+1} = a_m - 1$$

$$para\ n - 1\ iteraciones$$

Observamos que el coste C es lineal pues a mayor n mayor diferencia respecto $O(n^2)$ se notará. Finalmente concluimos $C \in O(n)$.

Con este resultado parece efectivo, en términos de tiempo, utilizar el NN. Para luego aplicar búsqueda local sobre la solución generada. De esta forma conseguiremos aproximarnos a la optimalidad en un tiempo razonable.

3.2. Hill Climbing

Se trata de un algoritmo de búsqueda local de mejora, que aplica transformaciones sobre una solución hasta que da con una solución inmejorable, esta solución mejorada podría ser la óptima o tratarse de una subóptima, en cuyo caso se trata de un mínimo local. Este algoritmo no exige que se cumpla la desigualdad triangular en el grafo.

Existen dos tipos de algoritmos HC:

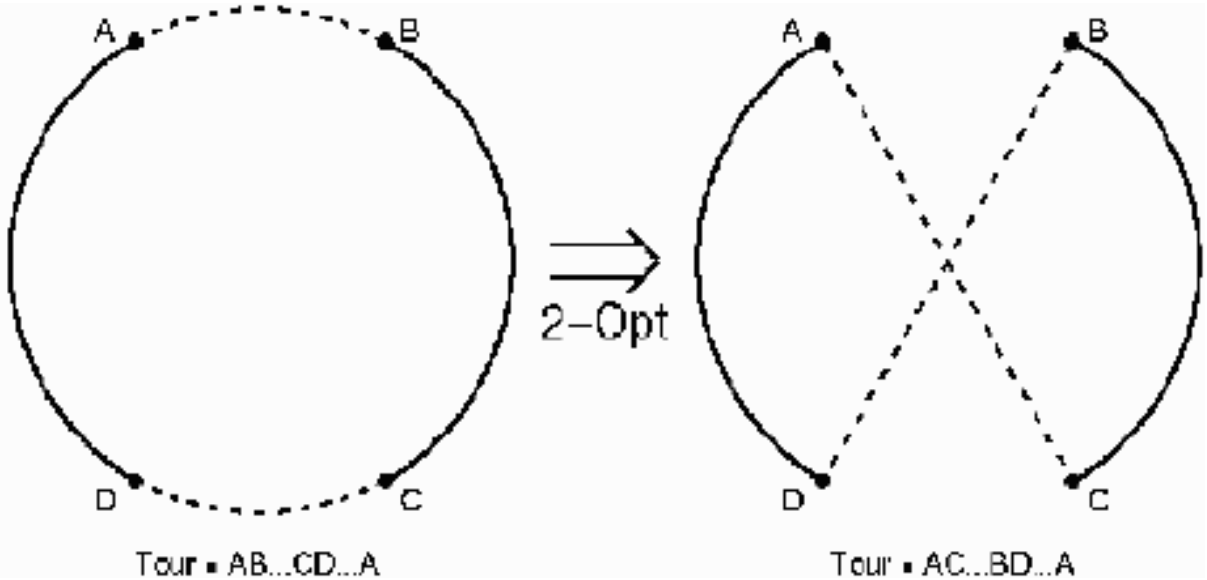
- Escalada simple
- Escalada por máxima pendiente

El sistema incorpora el HC de escalada simple, que consiste en la transformación de una solución aplicando un operador donde si la nueva solución transformada es superior a la anterior, se escala sobre la nueva. En caso contrario, vuelve a aplicar el operador en otra parte de la solución. Nótese que de esta forma, inmediatamente que el algoritmo ve una solución mejorada escala a partir de ella, y no genera todas las posibles permutaciones de soluciones transformadas y escala a partir de la mejor de ellas (como hace HC por máxima pendiente). HC de escalada simple consigue una exploración del espacio de soluciones mayor, ya que dependiendo de en qué par de vértices del camino se aplique el operador, se explorará una solución mejorada u otra, además consigue una mayor eficiencia temporal puesto que no ha de crear todas las permutaciones posibles.

Se ha decidido incorporar HC de escalada simple junto con una solución inicial del Nearest Neighbor -que para este, dependiendo del vértice inicial escogido el camino resultante será diferente-, así, se podrá hacer una mayor exploración del espacio de búsqueda, si lo ejecutamos sobre varias instancias diferentes del NN, en un menor coste temporal.

Los algoritmos de búsqueda local hacen uso de operadores que aplican sobre una solución concreta para posteriormente evaluar la mejora, existen numerosos operadores que se pueden

aplicar para problemas de categoría TSP. En particular para este sistema, **se ha incorporado el operador 2-Opt**.



Representación gráfica de aplicar el 2-Opt en un ciclo hamiltoniano.
Fuente: *Impact of grafting a 2-opt algorithm based local searcher into the genetic algorithm - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Exchange-step-of-2-opt-algorithm_fig1_262395470 [accessed 12 Dec 2024]*

El 2-Opt tiene como objetivo eliminar aquellas aristas para las cuales el camino se cruza a sí mismo. Es por esto, que utilizar este operador con HC de escalada por máxima pendiente parece poco efectivo, ya que eliminar aquellas aristas que se cruzan entre sí suele provocar una mejora del camino. Entonces, sobre una solución concreta, generar todas aquellas posibles permutaciones aplicando solamente una transformación 2-Opt sobre dos aristas cualesquiera -cómo hace HC de escalada por máxima pendiente-, para luego quedarse con la que más coste optimiza es ineficiente, pues prácticamente todos los cruces se tendrán eliminar para conseguir el óptimo, y se acabarán haciendo en iteraciones posteriores del HC. Por tanto, el coste de generar soluciones que optimizan pero que han sido descartadas se echa a perder. Dicho de otra forma, carece de sentido calcular cada una de las mejoras 2-Opt sobre el grafo para luego quedarme con la mejor, sabiendo que luego tendré que volverlas a calcular.

En general, para grafos de tamaños grandes, la mayoría de caminos subóptimos tienen aristas que se cruzan entre sí. Por esto, las operaciones K-Opt suelen utilizarse con frecuencia para resolver este tipo de problemas.

A continuación se presenta el pseudocódigo del operador 2-Opt en la Figura 2.

2-Opt (camino, vértice1, vértice2)
caminoResultante = θ caminoResultante.añadir(camino entre vérticePrincipio hasta vértice1) caminoResultante.añadir(camio reverso entre vértice1+1 hasta vértice2) caminoResultante.añadir(camino entre vértice2+1 hasta vérticePrincipio) devuelve caminoResultante

Figura 4: 2-Opt operación

Observamos que el coste de la operación 2-Opt es $O(n)$ puesto que recorre todos los vértices del grafo mientras los añade al caminoResultante. Esto implica que si utilizamos HC, por cada transformación que hagamos a una solución, tendremos que asumir un coste $O(n)$. Jugando en nuestra contra si, en caso peor, para cada par de vértices de la solución debemos aplicar este operador y luego comprobar que mejora. Sin embargo, esto se puede optimizar como condición de aplicabilidad del operador que tiene coste $O(1)$, permitiéndonos solo aplicar la transformación cuando se nos garantice una mejora. Esta optimización da motivos para utilizar HC de escalada simple para este tipo de problemas puesto que el coste temporal del algoritmo se ha reducido sustancialmente.

Condición de aplicabilidad para 2-Opt (costeActual, vértice1, vértice2)
costeMejorado = costeActual costeMejorado = costeActual - costeArista(vértice1, vértice1+1) - costeArista(vértice2, vértice2+1) + costeArista(vértice1, vértice2) + costeArista(vértice1+1, vértice2+1)

Figura 5: Condición de aplicabilidad para 2-Opt Optimizado

3.3. 2-Aproximación

Un algoritmo de 2-Aproximación a TSP es un algoritmo que encuentra una aproximación a la solución óptima del problema como mucho 2 veces peor, o sea el ciclo hamiltoniano tiene como mucho el doble del coste que el ciclo hamiltoniano mínimo.

Para que ésta 2-Aproximación se cumpla (y en general cualquier algoritmo de aproximación para el problema de TSP), el grafo inicial de entrada (G) tiene que satisfacer esta precondición:

$$\forall \text{ A, B, C distancia(A, B)} \leq \text{distancia(A, C)} + \text{distancia(C, B)}$$

Donde A, B y C son 3 vértices cualesquiera de G, esta condición es llamada la **desigualdad triangular.**

Un algoritmo de 2-Aproximación para el problema de TSP para nuestro grafo de entrada G es el siguiente:

1. **Construir el Árbol de expansión mínima T** (o MST, por sus siglas en inglés) para G, que es un subgrafo conexo sin ciclos de G que contiene todos sus nodos de manera que la suma de pesos de las aristas del subgrafo sea mínima, por construcción este subgrafo será un árbol.

   $Coste\ Temporal\ \ --> O(n^2 log(n^2))$
2. **Cada arista de T (x-y) la transformamos en 2 aristas dirigidas con sentidos opuestos (x → y) (y → x) y luego calculamos un circuito euleriano en el grafo transformado**, que es un ciclo que pasa por todas las aristas del grafo exactamente una vez.

   $Coste\ Temporal\ \ --> \ \ O(n)$
3. **Convertimos el circuito euleriano en ciclo hamiltoniano** eliminando los nodos repetidos del circuito.

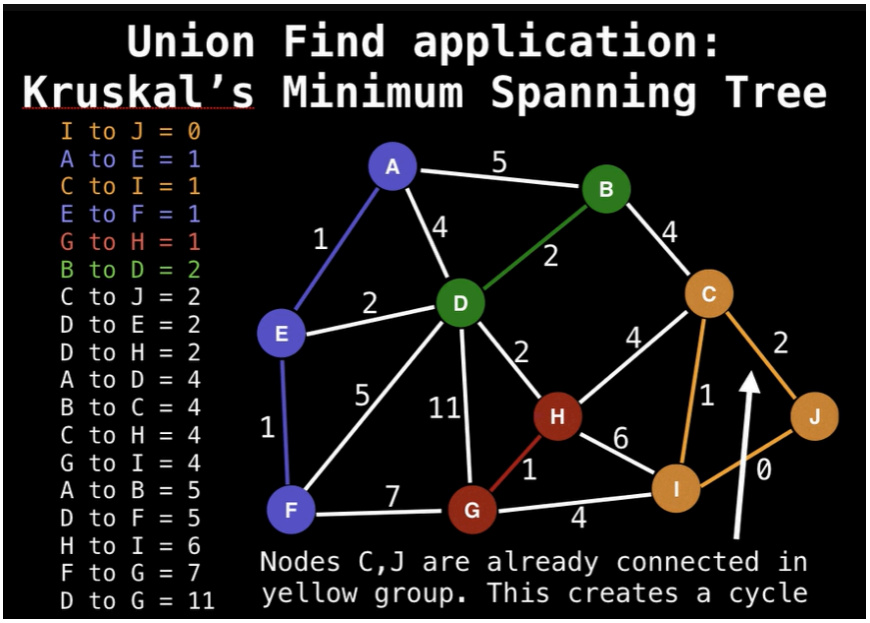
   $Coste\ Temporal\ \ --> \ \ O(n)$

$Coste\ Temporal\ Total\ \ --> O(n^2 log(n^2))$

3.3.1. MST con Kruskal

El algoritmo de Kruskal es un algoritmo ampliamente utilizado para calcular el MST de un grafo G, consiste en ir añadiendo a un árbol (inicialmente vacío) las aristas de menor peso de G de manera que no formen ciclos.

La manera optimizada de implementar el algoritmo de Kruskal que usaremos consiste en, ordenar primero todas las aristas del grafo en orden creciente, recorrerlas en ese orden y ir evaluando para cada arista si forma un ciclo en el árbol que hemos formado hasta el momento, Para esta comprobación de si forma un ciclo o no utilizaremos la estructura de datos eficiente de *Union Find* (o *Merge Find Set*, explicada en su apartado de Estructuras de Datos), evaluando para cada arista "nueva" (u,v) que queramos añadir al árbol si sus extremos (u y v) **no** pertenecen al mismo subconjunto disjunto, si sí pertenecieran al mismo conjunto disjunto se formaría un ciclo.



Representación gráfica de evaluar si una arista forma o no un ciclo usando Union Find en medio de una ejecución del algoritmo de Kruskal, en la imagen los distintos conjuntos disjuntos del árbol formado hasta el momento son representados por distintos colores, menos el blanco, que representa las aristas que aún no hemos evaluado, en la izquierda además vemos coloreadas las iteraciones hasta el momento. Se puede observar que la arista que conecta los vértices C y J, al pertenecer ambos vértices al mismo subconjunto disjunto, si la añadiéramos en el árbol, se formaría un ciclo.
Fuente: *https://youtu.be/JZBQLXgSGfs?t=239*

Este es el pseudocódigo del algoritmo de Kruskal:

MST con Kruskal (grafo G(V, E))
Aristas(|E|) ← el conjunto E de aristas de G ordenado crecientemente UF(|V|) ← inicializamos el Union Find con cada vértice en su propio conjunto disjunto T(|V|) ← inicializamos un grafo vacío para toda arista (i , j) en Aristas si UF.find(i) ≠ UF.find(j) UF.union(i , j) T ∪ (i, j) si no Aristas.eliminar(i, j) fin para todo retornar T

Figura 6: Algoritmo de Kruskal

El coste de ordenar es O(|E|log|E|), y como que el union y el find tienen un coste que a la práctica se puede considerar constante, entonces el coste de Kruskal quedaría como O(|E|log|E| + |E|), que es básicamente O(|E|log|E|) (en el problema general seria $O(n^2 log(n^2))$, donde n es el número de productos, ya que la matriz de costes representa a un grafo completo con los productos como vértices).

3.3.2. Circuito euleriano

Para buscar un circuito euleriano de una forma sencilla en el árbol anterior, primeramente tendríamos que duplicar las aristas del árbol, de manera que por cada arista del árbol creamos 2 aristas entre los mismos vértices con sentidos opuestos. Ahora, a partir de un vértice inicial arbitrario, vamos recorriendo el nuevo árbol con las aristas duplicadas en forma de DFS donde vamos marcando las aristas visitadas. Específicamente, para cada vértice que recorremos, primero exploramos (si tengo) los vértices a los cuales tengo una arista *de ida* y una *de vuelta* sin visitar, cuando los recorramos marcamos la arista *de ida* como visitada,

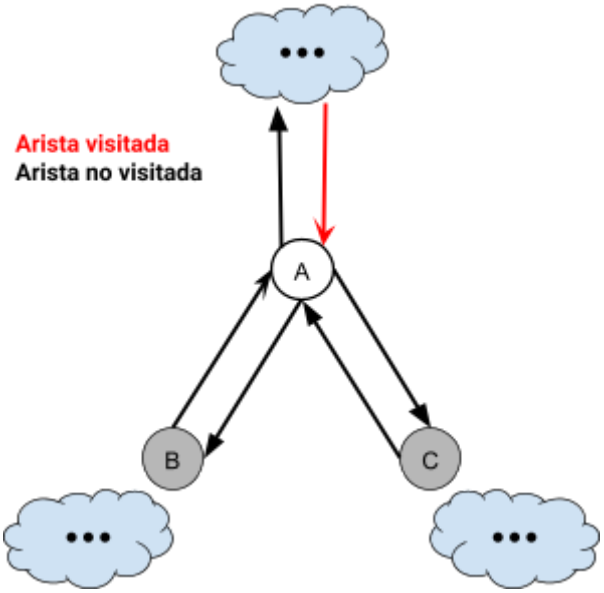


Figura 5: En la figura vemos un ejemplo del caso descrito, las "nubes" representan la continuación del grafo por los extremos, ya que en el dibujo se intenta mostrar un caso genérico de la recurrencia que se sigue para formar el ciclo euleriano. En este caso, la arista A visitará a los vértices **B y C**, ya que son los vértices a los que tiene una arista *de ida* y *de vuelta* sin visitar.

una vez ya recorridos todos los vértices a los que tengo una arista *de ida* y una *de vuelta* no visitados (tengo un camino para ir y volver aun no explorado), quiere decir que ya no me quedan caminos que "explorar" desde el vértice en el que estoy, así vamos volvemos por donde hemos venido anteriormente, que en este caso sería ir al vértice en el que tengo un vértice *de ida* no visitado pero uno *de vuelta* ya visitado (el *de vuelta* está visitado porque es por donde "vine anteriormente").

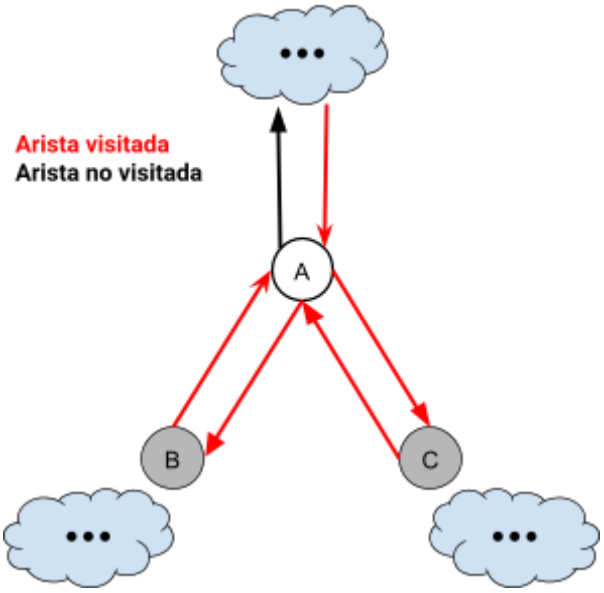


Figura 6: Ahora en el ejemplo vemos que desde la arista A ya hemos visitado a las aristas B y C y sus respectivos subárboles, una vez visitados, volveremos por el camino de vuelta hacia su A, y A volverá por la única arista de ida que tiene sin visitar (sabemos que "vuelve" por allí ya que la arista hacia A está visitada)

Este es el pseudocódigo:

Encontrar Circuito Euleriano En Árbol Duplicado(grafo G(V, E))
circuito ← nueva lista vacía pila ← nueva pila pila.push(0) mientras que la pila no esté vacía actual ← pila.pop() circuito.añadir(actual) encontrado ← false vérticeRetorno ← de momento no tengo Para todo vecino i de actual si tengo la arista (act → i) sin visitar si tengo la arista (act ← i) sin visitar pila.push(i) marco la arista (act → i) como visitada encontrado ← true

```
                        pila.push(i)
                        romper bucle 'para todo vecino i'
                    si no
                        vérticeRetorno ← i
        fin para todo

        si no encontrado
            pila.push(vérticeRetorno)

fin mientras

retornar circuito
```

Figura 7: Algoritmo para encontrar circuito euleriano en un árbol con las aristas duplicadas

El coste de hacer este recorrido es O(|V|) (O(n) si lo miramos en cuanto al número de productos), ya que recorremos cada arista una sola vez, y como sabemos, un árbol tiene |V|-1 aristas (en nuestro caso recorremos un árbol con las aristas duplicadas, pero asintóticamente eso no cambia nada).

3.3.3. Circuito euleriano → ciclo hamiltoniano

Ahora toca a partir del circuito euleriano pasar al ciclo hamiltoniano, nosotros intentamos implementar una manera voraz de eliminar los nodos repetidos del circuito euleriano para transformarlo en ciclo hamiltoniano, en la que para cada duplicado evaluábamos cómo afectaría al grafo eliminarlo o no, y si merecía la pena que se quede en el grafo.
Evidentemente este planteamiento del problema de manera voraz no nos dará el resultado óptimo, ya que eliminar un duplicado en concreto afecta al resto del circuito de manera no trivial. Para obtener el resultado óptimo tendríamos que considerar todas las permutaciones posibles de eliminar los duplicados, y eso tiene un coste exponencial.
Por eso, experimentando con nuestro algoritmo voraz, hemos visto que en la práctica no da muy buenos resultados, y hemos verificado que eliminar los vértices repetidos del nodo de manera aleatoria es mucho más efectivo en la absoluta mayoría de los casos, así que eso es lo que hemos hecho. A partir de un nodo aleatorio escogido en el circuito euleriano, hemos

recorrido el grafo eliminando los duplicados (quedándonos con la primera instancia del vértice en el circuito), lo cual nos da como resultado un ciclo euleriano, Como este recorrido aleatorio solamente tiene coste lineal, lo hacemos un total de 3 veces y nos quedamos con el ciclo hamiltoniano con coste mínimo.

Este sería el pseudocódigo:

De Euleriano A Hamiltoniano (circuito C)
random ← generamos un nodo random caminoHamiltoniano ← vacío mientras hayamos recorrido todo nodo 'n' del circuito empezando por 'random' si el nodo n no está visitado marcamos n como visitado caminoHamiltoniano.añadir(n) fin mientras

Figura 8: Circuito euleriano a ciclo hamiltoniano

El coste seria O(|V|) (O(n) si lo miramos en cuanto al número de productos), ya que recorremos 3 veces el circuito euleriano, que tiene tamaño O(|E|) sobre el grafo que recorre, y, por el mismo argumento que hemos dado para justificar el coste en el apartado **3.3.2.**, sabemos que ese número de aristas es O(|V|) respecto al "grafo" original.

4. Solución Óptima

Para saber cómo de bien funcionan nuestros 3 algoritmos de aproximación, usaremos un algoritmo extra que nos retorne la solución óptima de TSP, hay muchas maneras de conseguir esta solución óptima, nosotros usaremos una de las más simples (y costosa, exponencialmente costosa), que es la del backtracking con poda, la cual solamente nos funcionará para grafos muy pequeños, del orden de 10 ~ 20 nodos, para grafos mayores sería inviable el coste temporal de encontrar una solución óptima con este algoritmo.

La poda que usaremos en el backtracking consiste en, en el proceso de construir un camino, comprobar que el coste del camino construido hasta el momento es menor al camino con coste mínimo encontrado hasta el momento, si no se cumple esto (al no tener pesos negativos), ya podemos descartar el camino que se está construyendo.

Este sería el pseudocódigo:

```
SoluciónÓptima (nodo_actual, camino_actual, coste_actual)

si el camino_actual tiene la misma longitud que |V|
        coste_total = coste_actual + coste entre nodo_actual y camino_actual[nodo_inicial]

        si coste_total < coste_minimo
                coste_mínimo = coste_total
                camino_minimo = camino_actual

        retornar

si no
        para todo nodo i en el grafo
                si i no está en camino_actual
                        coste_nuevo = coste_actual + coste entre nodo_actual y i
                        si nuevo_coste < minimo_coste
                                añadir nodo i a camino_actual
                                SoluciónÓptima(nodo i, camino_actiual, coste_nuevo
                                quitar nodo i de camino_actual
```

Figura 9: Backtracking para la solución óptima

5. Bibliografía

[1] https://www.ijeat.org/wp-content/uploads/papers/v4i6/F4173084615.pdf

[2] I. Akhmetov and A. Pak, "TSP review: performance comparison of the well-known methods on a standardized dataset," 2023 19th International Asian School-Seminar on Optimization Problems of Complex Systems (OPCS), Novosibirsk, Moscow, Russian Federation, 2023, pp. 4-9, doi: 10.1109/OPCS59592.2023.10275749.

[3] https://en.wikipedia.org/wiki/2-opt

[4] https://www.youtube.com/watch?v=GiDsjIBOVoA&t=419s&pp=ugMICgJlcxABGAHKBQN0c3A%3D

[5] https://www.cs.upc.edu/%7Emjserna/docencia/grauA/T24/5-A-GEI-mst-h.pdf