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.

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 1: 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**.

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 2: 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 3: Condición de aplicabilidad para 2-Opt Optimizado

4. 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