I. Diseño del Experimento

1. Algoritmo de dijkstra:

Trataremos de explicar brevemente el algoritmo de Dijkstra y del uso en el lenguaje R:

Dado un grafo conexo, dirigido o no dirigido, con distancias no negativas; el algoritmo de Dijkstra empieza por marcar el nodo fuente e inicializar a 0 o a infinito las distancias acumuladas de cada nodo. Posteriormente itera, mientras queden nodos sin marcar, seleccionando en cada etapa el arco con la suma más pequeña de la distancia acumulada por uno de los nodos marcados más la distancia del arco que lo une a uno de los nodos no marcados, guardando este último nodo como marcado y actualizando su distancia acumulada con la suma antes calculada. Al final del proceso tendremos un árbol de expansión de camino más corto. La implementación del algoritmo tal y como está escrita en el **pseudocódigo** no es casual y facilita portarlo a R. Existe varias implementaciones del algoritmo al margen de la original de Dijkstra. De hecho, la que utilizaremos en el paquete **optrees** modifica un poco su funcionamiento para adaptarlo a nuestras necesidades. Por otra parte, es necesario, eso sí, indicar si el grafo que introducimos en la función es o no dirigido, pues en este segundo caso deberemos realizar el paso previo de duplicar los arcos. El resto de datos del grafo de entrada lo constituyen el conjunto de nodos y arcos, los cuales se introducen en R de la misma forma que hemos visto hasta ahora, por lo demás, las instrucciones del algoritmo de Dijkstra se pueden trasladar directamente manteniendo el uso de un único bucle, lo que permite evitar aumentar la complejidad del problema. La propuesta original de Dijkstra funciona en un tiempo computacional teórico de $O(|V|^2)$. Con el tiempo se han desarrollado mejores implementaciones que requieren una cola de prioridad y reducen la complejidad del algoritmo, pero complican portal el código a R. En nuestro caso, aunque con diferencias, mantenemos un formato muy similar al modelo original de Dijkstra, que es de por sí lo suficientemente rápido como para obtener resultados por debajo del segundo en grafos con cientos de nodos y miles de arcos.

I-A. Optrees

Este paquete en R, encuentra árboles óptimos ponderados. En particular, este paquete proporciona herramienta de resolución de problemas de árbol de expansión de costo mínimo, problemas de arborescencia de costo mínimo, problemas de árbol de ruta más corto y problema de árbol de corte mínimo.

A continuación te mostramos el Pseudocódigo:

Algorithm 1 Dijkstra
Entrada: $G = (V, A); D, s \in V$
Salida: $G^T = (V^T, A^T); W$

$A^T \leftarrow \emptyset$
$V^T \leftarrow \{s\}$
$W_{1,|V|} \leftarrow 0$
while $|V^T| < |V|$ **do**
    seleccionar arcos $(i, j) \in A$ con $i \in V^T$ y $j \in VV^T$
    elegir un arco (i, j) de los anteriores tal que $d_{ij} + W_i$ sea mínimo
    $A^T \leftarrow A^T \cup \{(i, j)\}$
    $V^T \leftarrow V^T \cup \{j\}$
    $W_j \leftarrow d_{ij} + W_j$
end while

2. Algoritmo de Bellman-Ford:

Este segundo algoritmo, para el problema del árbol de camino más corto es el conocido como algoritmo de Bellman-Floyd. El algoritmo empieza inicializando a 0 la distancia asociada a la fuente y a infinito las distancias acumuladas del resto de nodos. Posteriormente, lleva a cabo el proceso de relajación iterando en todos los nodos distintos de la fuente, comprobando en cada arco que sale de él si la distancia hacia otro nodo es mayor que la suma de la distancia acumulada del primero más el peso del arco que los une, en caso afirmativo establece este resultado como la distancia acumulada por el nodo de llegada y guarda el nodo de partida como su predecesor. Al final termina comprobando si se han encontrado ciclos negativos, revisando arco por arco si se puede reducir la distancia acumulada del nodo de llegada. Si no se topa con uno, el resultado final es un árbol de expansión del camino más corto. El proceso de programar el algoritmo de Bellman-Floyd en R es equivalente al pseudocódigo aquí desarrollado, aunque con ligeros añadidos. En este caso, también es necesario introducir un parámetro de entrada que permita señalar si estamos ante un grafo dirigido o no, para duplicar los arcos ante eseta segunda situación, pero al final es necesario incorporar un bucle adicional que reconstruya el árbol del camino más corto a partir del conjunto de nodos y los predecesores que hemos ido almacenando durante el proceso de relajación. Este último bucle ralentiza un poco nuestra función, tiene una complejidad de $O(|V||A|)$. Es, por tanto, más lento que el de Dijkstra, siendo este el precio a pagar por permitir trabajar con pesos negativos.

A continuación te mostramos el Pseudocódigo:

Algorithm 2 Bellman-Ford
Entrada: $G = (V, A); D; s \in V$
Salida: $G^T = (V^T, A^T); W$

```
A^T ← ∅
W_{1:|V|} ← ∞; W_s ← 0
P ← ∅
for cada nodo i ∈ V{s} do
    for cada arco (i, j) ∈ A do
        if W_j > W_i + d_ij then
            W_j ← W_i + d_ij
            P_j ← i
        end if
    end for
end for
for cada arco (i, j) ∈ A do
    if W_j > W_i + d_ij then
        No hay solución
    end if
end for
for cada nodo j ∈ V{s} do
    i ← P_j
    seleccionar arco (i, j) de A
    A^T ← A^T ∪ {(i, j)}
end for
V^T ← V
```

3. Algoritmo de Gusfield:

La principal ventaja del algoritmo de Gusfield es, precisamente, evitar el requisito de comprimir nodos. El algoritmo arranca construyendo un arbol con el primer nodo como nodo único, e itera posteriormente añadiendo cada vez uno nuevo de acuerdo al orden $2, 3, ..., |V|$. En cada iteración la duda radica en escoger a qué nodo del árbol se une cada nuevo nodo k cuando existe más de una posibilidad.

Para determinarlo repetimos el siguiente proceso:

- Buscamos el arco (i, j) de menor peso dentro del árbol. Este arco tendrá el valor del corte mínimo entre los nodos i y j en el grafo original.
- Borramos dicho arco del árbol formando dos componentes, una con el nodo $i(T_1)$ y otra con el nodo $j(T_2)$. Con ellos sabremos también los nodos que pertenecen a la misma componente que $i(S_1)$ y los que pertenecen a la musma componente que $j(S_2)$ en el corte $i - j$ del grafo original.
- Si el nodo k pertenece a la componente S_1 en el grafo original, entonces fijamos como nuevo árbol a comprobar la componente T_1 del árbol. En caso contrario fijamos como nuevo árbol a comprobar la componente T_2 del árbol.
- Repetimos los pasos anteriores hasta que nos quedemos conn un árbol con un único nodo.

Este será el nodo del árbol al que tenemos que unir el nuevo nodo k. El peso del arco que los una será igual al corte mínimo entre los dos nodos en el grafo original.

Todo este proceso se realiza hasta completar un árbol con todos los nodos del grafo y un nuevo conjunto de arcos. Este será un árbol des corte minimal en el que cada arco (i, j) se corresponda con la capacidad del corte $i - j$ mínimo en el grafo original.

A continuación te mostramos el Pseudocódigo:

Algorithm 3 Gusfield
Entrada: $G = (V, A); Z$
Salida: $G^{T'} = (V^{T'}, A^{T'}); Z^T$

```
V^T ← {1}
A^T ← ∅
Z^T_{|V|X|V|} ← ∞
for cada nodo i ∈ V1 do
    V_k^T ← V^T; A_k^T ← A^T
    while |V_k^T| > 1 do
        borrar un arco (i, j) ∈ A_k^T cuyo a_{i,j} sea mínimo
        determinar componentes T_1 y T_2 de V_k^T y A_k^T tal que i ∈ T_1 y j ∈ T_2
        obtener corte mínimo entre i y j en el grafo G = (V, A) original
        determinar componentes S_1 y S_2 de V^T y A^T tal que i ∈ S_1 y j ∈ S_2
        if nodo i ∈ S_1 then
            V_k^T ← nodos de la componente T_1
            A_k^T ← arcos de la componente T_1
        else
            V_k^T ← nodos de la componente T_2
            A_k^T ← arcos de la componente T_2
        end if
    end while
    añadir arco (i, k), con k ∈ V_k^T, al árbol A^T
    obtener corte mínimo entre i y k en el grafo G = (V, A) original
    Z_ik^T ← capacidad del corte mínimo i − k
    V^T ← V^T ∪ {i}
end for
```