TEORÍA DE ALGORITMOS 1

Metodología greedy

Por: ING. VÍCTOR DANIEL PODBEREZSKI
vpodberezski@fi.uba.ar

1 Introducción

Cuando nos enfrentamos a un problema complejo una práctica que suele buscarse es descomponer el mismo en partes más pequeñas. De esa forma enfrentaremos subproblemas menores que podemos resolver en forma más sencilla. Una consecuencia de esa separación es la necesidad de generar un mecanismo para interconectar y jerarquizar las partes. La solución de un subproblema puede ser requerido para atacar a otro. El tratamiento del conjunto de los subproblemas nos dará la solución global del problema original.

La mayoría de los paradigmas de resolución de problemas enfrentan esta separación e interconexión de diferentes maneras. Los conocidos como algoritmos codiciosos (Greedy) son tal vez los más fáciles de producir entre ellos. Esta familia de algoritmos se utilizan para resolver problemas de optimización. Intentan por lo tanto maximizar o minimizar alguna cantidad durante su ejecución. El algoritmo realiza una división jerárquica en subproblemas cuya resolución habilita un nuevo subproblema a resolver. El proceso es en general iterativo. En cada subproblema se aplica un criterio de elección de solución que se basa únicamente en la situación local y conocida. Trata de maximizar el objetivo con la información que tiene disponible. Es por este motivo que recibe el nombre. También se lo suele llamar “avaro” o “miope”. No mira la situación global, sino que elige localmente lo que considera mejor según un criterio preestablecido.

Si bien es prácticamente posible crear un algoritmo greedy para cualquier problema de optimización, no siempre la solución obtenida será óptima. Se requiere que el problema cumpla con ciertas características. Y aun cumpliendolas si seleccionamos criterios incorrectos nuestra solución puede ser imperfecta. La primera característica que debe

cumplir el problema la llamaremos **subestructura óptima** (optimal substructure). Implica que la solución al problema contiene dentro de sí la solución óptima de sus sus subproblemas. Es esta propiedad la que justamente nos permite la separación del problema en partes. Por otro lado. Tendremos generalmente una división jerárquica de subproblemas donde la resolución de uno de ellos permite seleccionar la solución óptima del siguiente. Sin embargo un algoritmo greedy no ahondará en todos los subproblemas posibles. sino que realizará una **elección codiciosa** (greedy choice) y en vez de proseguir por todas las posibles subproblemas profundizará solo en uno de ellos. Si el problema admite esa elección entonces se llegará a una solución óptima global. Esta elección además debe ser factible (que cumple con las restricciones del problema) e irrevocable (una vez tomada no permite ser removida en posteriores elecciones del algoritmo).

Veremos a continuación una selección de algunos problemas para los que desarrollaremos un algoritmo de tipo greedy como estrategia de resolución. Para cada uno de ellos probaremos su optimalidad y mediremos su complejidad.

2 El problema de la mochila y los elementos divisibles.

Este problema, que se conoce en la bibliografía como mochila fraccionable (fractional knapsack), es un buen punto de partida para analizar un algoritmo greedy. En este, se nos plantea la necesidad de seleccionar un conjunto de elementos para maximizar alguna magnitud disponible (monetaria u otra). Cada elemento cuenta con una cantidad máxima y un valor que representa el incremento que obtenemos al seleccionarlo. Estos tienen la capacidad de fraccionarse en partes, sin restricción de tamaño. El beneficio por una porción de un elemento es linealmente proporcional a la cantidad desprendida. En contra del deseo de llevarse todo, contamos con una restricción (peso, volumen u otra medida) de la que no podemos excedernos. A medida que seleccionamos una cantidad de un elemento, ese espacio disponible se reduce. Se visualiza a esa restricción como una mochila que se va llenando con nuestras elecciones y de allí el nombre del problema.

Lo enunciaremos formalmente y luego daremos un ejemplo del mismo

Mochila fraccionaria

Contamos con una mochila con una capacidad de K kilos y queremos introducir dentro de ellas un subconjunto del conjunto E de "n" elementos con el objetivo de maximizar la ganancia. Cada elemento i tiene se encuentra en una cantidad de c_i kilos y un valor de v_i. Los elementos pueden ser fraccionados en la cantidad que consideremos necesaria.

Somos los dueños de un próspero negocio de transportes interplanetario que realiza una ruta entre la Tierra y Marte. Contamos con un carguero que puede transportar hasta 760 toneladas de productos a granel. En el próximo viaje puede llevar hasta 250 toneladas de agua por 300 PS (pesos solares), 200 toneladas de trigo por 400 PS, 320 toneladas de oxígeno por 280 PS, 200 toneladas de aceite por 200 PS y 100 toneladas de aluminio por 300 PS. Debemos determinar que transportar el próximo viaje para maximizar las ganancias.

Si queremos encontrar un algoritmo de tipo greedy que de solución al problema planteado debemos ver la forma de jerarquizar los subproblemas y definir el criterio de elección en cada uno de ellos. Vemos que cualquier estrategia iterativa debería ir seleccionando algún elemento en alguna cantidad para ingresar en la mochila. Por cada elección disminuye la capacidad libre y la cantidad de los elementos disponibles. Cada elección define un nuevo escenario donde se abren nuevas elecciones. Hasta que al finalizar no hay lugar disponible y se consigue un valor acumulado como ganancia.

Podemos representar la ganancia como $G = \sum_{i \in E} \frac{v_i}{c_i} \cdot u_i$ donde u_i es la cantidad del elemento i seleccionado ($c_i \leq u_i$) y el cociente corresponde al valor por unidad del elemento. El objetivo es obtener Max(G) el valor máximo de ganancia. El único parámetro en el que tenemos ingerencia es la cantidad que elegimos de cada elemento. La suma de estas no pueden superar la capacidad de la mochila $K \leq \sum_{i \in E} u_i$. Nuestra elección codiciosa tiene que determinar justamente la cantidad de cada elemento a elegir y en qué orden realizarlo.

Una primera decisión corresponde a si al seleccionar un elemento tratamos de ingresar la mayor cantidad posible de este o si lo fraccionamos en cierto valor relacionado con el tamaño de la mochila u otro criterio. En ciertos casos esto último puede tener sentido. Por

ejemplo, si existe alguna restricción de diferencia de cantidades entre elementos. Pero no es el caso aquí.

El siguiente dilema es cómo elegir el próximo elemento a incluir. Puede ser alguno de los siguientes criterios: mayor cantidad disponible, menor cantidad disponible, mayor valor del producto, menor valor del producto, mayor valor por unidad del producto, menor valor por unidad del producto, … entre otros. Este caso resulta un problema simple y posiblemente el criterio a elegir no sea difícil de encontrar. De todas formas analizaremos varios y ellos y encontraremos un contraejemplo para los que no funcionan como esperamos. Recordar que la solución propuesta debe funcionar bien para cualquier instancia del problema. Finalmente para aquel que parece ser correcto probaremos matemáticamente que lo es.

Si el criterio de elección fuese “mayor cantidad disponible”. Podemos encontrar un ejemplo donde este falla. Imaginemos que tenemos una capacidad de 5 en la mochila y tenemos un producto “A” con cantidad 5 y valor 1. Por otro lado tenemos el producto “B” con cantidad 1 y valor 5. Con nuestro criterio llenaríamos la mochila con “A” consiguiendo una ganancia de “1”. Sin embargo si ingresamos el producto “B” y luego ⅘ partes de “A” tendríamos una ganancia mayor.

Tal vez la solución entonces es seleccionar el de “menor cantidad disponible”? La respuesta es negativa. En el ejemplo anterior si ahora los valores de “A” y “B” son 10 y 1 respectivamente. Terminariamos seleccionado al producto “B” y ⅘ partes de “A” nuevamente obteniendo un valor menor de simplemente seleccionar todo el producto “A” para llenar la mochila.

Podemos entonces pensar que la solución es ingresar primero aquellos con “mayor valor”. De nuevo podemos encontrar un contraejemplo. Contamos con la misma mochila y ahora tenemos a los productos “A” con valor 5 y cantidad 5 y por otro lado el producto “B” con valor 4 y cantidad 1. De acuerdo a nuestra propuesta de elección solo ingresariamos el producto “A” y obtenemos una ganancia de 5. Pero claramente conviene poner “B” y ⅘ de “A”.

En este punto podemos también probar utilizando el resto de los criterios, con todos ellos encontraremos un contraejemplo excepto para el de “mayor valor por unidad del producto”. ¿Es tal vez esté el buscado?

En ocasiones encontrar un contraejemplo se vuelve complicado y es necesario analizar diferentes escenarios y casos especiales. Muchas propuestas de algoritmos greedy que se quedan en un análisis de este tipo suelen fallan porque - tal vez por el deseo que funcione bien - quedan casos sin analizar y todo parece funcionar bien. Es por eso que llegado a un punto donde todo parece funcionar bien, tenemos que probarlo formalmente. Y esta parte suele ser la más compleja dentro de esta metodología.

Antes de continuar pasemos a pseudocódigo nuestro algoritmo con el criterio a analizar

Completar mochila de forma greedy
lugarMochila = K Mientras lugarMochila > 0 y elementos disponibles Sea elem el elemento disponible de mayor v_i / c_i Ingresar en la mochila max(c_{elem},lugarMochila) Descontar max(c_{elem},lugarMochila) de lugarMochila Quitar elem de elementos disponibles

Para nuestro carguero espacial vemos que el valor por unidad de los elementos disponibles es el siguiente. Agua → 300/250 = 1,2. Trigo → 2. Oxígeno → 0,875. Aceite → 1. Aluminio → 3. Por lo tanto nuestro algoritmo seleccionará de acuerdo a estos valores de mayor a menor. En nuestro carguero quedarán: 100 toneladas de aluminio, 200 toneladas de trigo. 250 Toneladas de Agua. 200 Toneladas de Aceite y 10 Toneladas de Oxígeno. Teniendo los depósitos llenos con 760 toneladas y una ganancia de 1287,5

¿Es nuestro algoritmo óptimo? Intentaremos demostrarlo por el absurdo. Llamamos OPT a una solución óptima del problema. Llamamos S a la solución obtenida mediante nuestro algoritmo. Consideramos a los elementos E= {e_1, e_2, e_3, .., e_n}, ordenados con el criterio de elección del algoritmo greedy. Vamos a pedir como extra que no existan dos elementos que tengan el mismo valor por unidad. Las cantidades seleccionadas en OPT las podemos representar como un vector (o_1, o_2, … , o_n) donde la posición i tiene la cantidad incluida del elemento e_i . De igual manera las cantidades seleccionadas por la solución S las representamos en un vector (s_1, s_2, … , s_n)

Supongamos que la solución encontrada por el algoritmo no es óptima. Es decir que $\sum_{i \in E} \frac{v_i}{c_i} \cdot o_i > \sum_{i \in E} \frac{v_i}{c_i} \cdot s_i$ entonces tiene que existir al menos un elemento con cantidades distintas entre ambas soluciones. Tomemos x el elemento de mayor valor por unidad que difieren. Como la elección greedy toma la máxima cantidad posible en orden descendiente entonces $o_x > s_x$. Esa cantidad de diferencia tiene que estar distribuida entre los siguientes elementos de E. Llamemos $\delta = o_x - s_x$. Supongamos que el elemento inmediatamente posterior x+1 tiene esa diferencia (cualquier otra opción nos dará un valor de ganancia menor). Vemos que si en OPT movemos δ del elemento x+1 al elemento x, tenemos una diferencia positiva en la ganancia final de $\frac{v_x}{c_s} \cdot \delta - \frac{v_{x+1}}{c_{s+1}} \cdot \delta$ (puesto que $\frac{v_x}{c_s} > \frac{v_{x+1}}{c_{s+1}}$). Por lo que nuestra solución óptima se podría mejorar. Lo que corresponde a una contradicción de que OPT es una solución óptima y un absurdo. Por lo tanto, la solución greedy encontrada tiene que ser igual a la óptima. Por otra lado, esta demostración también nos dice que si todos los elementos tienen diferentes valores por unidad, entonces existe una única solución óptima y equivale a la obtenida por la solución greedy.

¿Qué pasa si hay más de un elemento con el mismo valor por unidad? En ese caso elegir una unidad de un elemento o el otro nos aporta la misma ganancia. Podemos a modo de análisis - y en ausencia de restricciones extras - tratarlos como él mismo elemento con la suma de las cantidades como cantidad disponible y el valor unitario común a todos ellos. Con esa transformación la demostración de optimalidad se mantiene. Vamos a un caso simple. Supongamos que existen dos únicamente dos elementos x, y con el mismo valor por unidad $\frac{v_x}{c_s} = \frac{v_y}{c_y}$. Llamaremos al nuevo elemento z que reemplazará a los anteriores con cantidad $c_z = c_x + c_y$ y valor de $\frac{v_x}{c_s}$. Buscamos la solución óptima. Si la misma incluye nada o la totalidad de c_z , entonces aún hay una única solución óptima. Si por el contrario la solución óptima transformada (con el elemento z en lugar de x e y) contiene un valor $o_z < c_z$ y si realmente cada elemento es infinitamente fraccionable, entonces se pueden construir infinitas soluciones óptimas con los elementos originales. Podemos componer esta cantidad como $o_x+o_y=o_z$. Y claramente esta igualdad se puede lograr de infinitas maneras con números fraccionarios. En la práctica utilizando el algoritmo greedy no tenemos que generar elementos ficticios. Simplemente se procesarán los elementos en el orden que

quedaron luego del ordenamiento. En ese sentido ante el mismo orden el algoritmo encontrará siempre la misma solución.

Ya comprobamos que nuestro algoritmo de tipo greedy es óptimo. Veamos ahora si es eficiente. Para eso tenemos que analizar su complejidad temporal y espacial con respecto al tamaño del input. En primer lugar tenemos que calcular el valor por unidad de cada elemento. Procesar y almacenar esa información requiere $\mathbf{O}(n)$. El proceso requiere recorrer los elementos teniendo en cuenta el criterio de elección. Debemos ordenarlos para eso y podemos usar un algoritmo con complejidad $\mathbf{O}(n \log n)$. Por último debemos recorrer en el peor de los casos todos los elementos para ir completando la mochila en $\mathbf{O}(n)$. Los elementos seleccionados con sus cantidades requieren $\mathbf{O}(n)$ de espacio de almacenamiento. Por lo tanto el algoritmo tiene una complejidad temporal de $\mathbf{O}(n \log n)$ y una espacial de $\mathbf{O}(n)$.

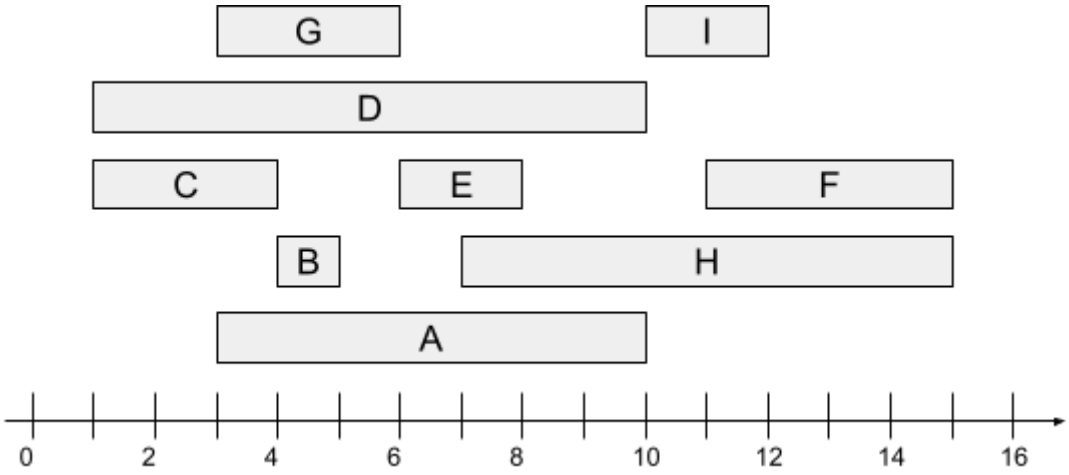
3 Selección de tareas

El siguiente problema con el que vamos a trabajar recibe dos nombres en la bibliografía. En algunos papers y libros se lo llama “Problema de selección de tareas” (activity selection problem) y en otras “Problema de maximización de programación de intervalos (interval scheduling maximization problem o “ISMP” por sus siglas). Como se puede apreciar corresponde a un problema de optimización donde se espera maximizar la cantidad de actividades a seleccionar. Una actividad (que puede ser una conferencia, una proceso productivo, una excursión) corresponde a un elemento que tiene una duración determinada. Tiene una fecha de inicio y una fecha de finalización. Como alternativa se puede representar por una fecha de inicio y una duración. Para llevar a cabo la actividad contamos con un único recurso (una sala, una máquina, un guía de turismo) que puede tomar una actividad por vez. Diremos que dos actividades son incompatibles si se solapan temporalmente en algún momento de su periodo de ejecución. El recurso no puede tener asignado dos o más actividades incompatibles. El problema a resolver es, por lo tanto, dado un conjunto de actividades, seleccionar la mayor cantidad de ellas compatibles entre sí.

Antes de analizarlo, lo enunciaremos formalmente y daremos un ejemplo del mismo

Selección de actividades
Contamos con un conjunto de “n” actividades entre las que se puede optar por realizar. Cada actividad x tiene una fecha de inicio l_x y una fecha de finalización f_x. Seleccionar el mayor subconjunto posible de ellas compatibles entre sí.

Un emprendedor ha comprado una lancha para realizar paseos de pesca en un lago. En su primera temporada vacacional ha creado un convenio con varias agencias de turismo. Ellas le pagan un valor fijo por exclusión realizada. Cada excursión tiene un día de inicio y de finalización. Para la próxima quincena desea aceptar aquellas que le permitan realizar más excursiones. Las opciones son (identificador, inicio,fin): (A,5,10), (B,4,5), (C,1,4), (D,1,10), (E,6,8), (F,11,15), (G,3,6), (H,7,15), (I,10,12)



Comencemos con el análisis del problema. En primer lugar vemos que podemos crear un proceso iterativo que en cada etapa seleccionando una actividad a realizar. Esto dará lugar a un subproblema resultante que excluya todos las otras actividades incompatibles con la recién agregada. El proceso terminará cuando no queden actividades disponibles. Supongamos que conocemos una actividad “X” que pertenece a una solución óptima. Vemos que podemos dividir la resolución del problema en dos subproblemas. Por un lado aquellos problemas compatibles y que comienzan luego que finaliza “X”. Por otro lado a aquellos problemas compatibles y que finalizan antes que “X”. La resolución óptima de estos dos subproblemas nos llevará a la solución óptima global. Allí vemos que existe una subestructura óptima en este problema. Para el criterio de elección codiciosa se pueden

analizar varias alternativas, entre ellas: mayor o menor duración, mayor o menor cantidad de incompatibilidades, mayor o menor fecha de inicio, mayor o menor fecha de finalización. Parece razonable desestimar aquellas donde se habla de mayor duración y mayor incompatibilidades. Si la idea es maximizar la cantidad seleccionada actividades compatibles entonces no parece una buena idea estas opciones (si el lector no está convencido, puede buscar contraejemplos rápidamente). Analizaremos a continuación el resto de ellas.

Veamos si el criterio de elección corresponde a “menor duración”. Parece una buena idea dado que si el tiempo de ocupación es pequeño nos da más lugar para introducir otras actividades. Pero en el siguiente contraejemplo se demuestra que puede traernos resultados no óptimos. Contamos con 3 actividades con tiempos (A,1,8), (B,9,15), (C,7,10). Sus duraciones son 7, 6 y 3 respectivamente. La primera actividad elegida sería la “C” y esta es incompatible con las dos restantes. Nos queda como solución “C” Sin embargo la solución óptima corresponde a “A” y “B”.

Como siguiente criterio de elección probaremos “menor cantidad de incompatibilidades”. Puede ser una buena idea seleccionar en primer lugar a aquellas actividades que eliminan la menor cantidad posible de otras. Aunque como muestra el siguiente contraejemplo esto tampoco nos lleva siempre a la solución deseada. Contamos con 11 actividades con tiempos (A,1,3), (B,2,5), (C,2,5), (D,2,5), (E,4,7), (F,6,9), (G,8,11), (H,10,13), (I,10,13), (J,10,13), (K,11,15). El que tiene menos incompatibilidades es la actividad F (con E y G). Por lo que es el primero seleccionado. Quitando las incompatilidades ahora todos los que quedan tienen las misma incompatibilidades restantes (3 cada una). Por ese motivo podemos seleccionar cualquiera de ellas, por ejemplo A. Restan luego de quitar las incompatibles H,I,J y K. Nuevamente cualquiera de ellas tienen el mismo número de incompatibilidades. Por lo que elegimos uno de ellos: el H. Nos queda la selección de las 3 actividades A,F y H. Por otra parte tenemos la posibilidad de seleccionar 4 actividades con A,E,G y K. Por lo tanto este criterio tampoco sirve.

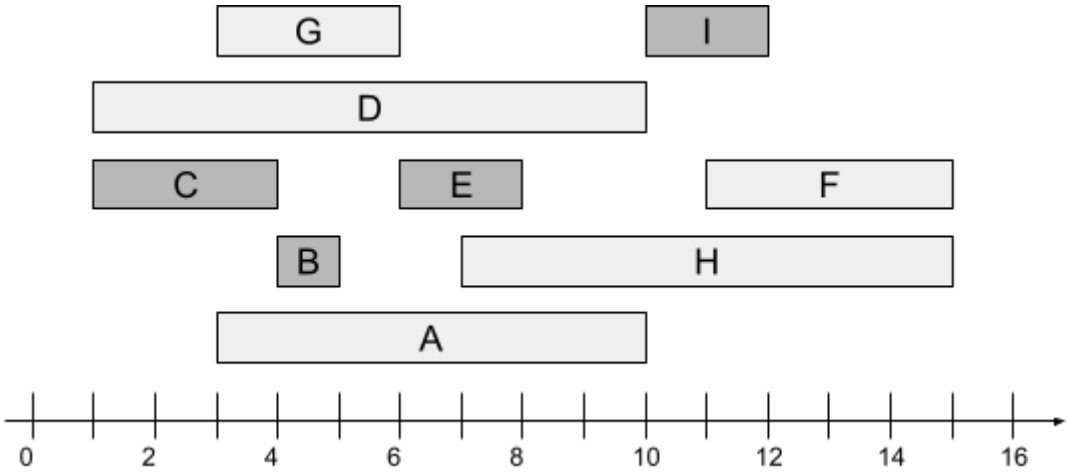
Tal vez sirva seleccionar primero a los que inician primero? Es fácil ver que no. Por ejemplo si tenemos 3 actividades la primera (A,1,10) inicia antes que (B,2,5) y (C,6,12). Al seleccionarla dejamos afuera a las otras dos por ser incompatible con ellas. Sin embargo podríamos seleccionar B y C consiguiendo una solución mejor. Y la opción primero el que

finaliza primero? Podemos buscar arduamente un contraejemplo y no encontramos. Parece un buen candidato para la elección greedy. Antes de continuar, pasemos en limpio el algoritmo mediante pseudocódigo.

Seleccionar actividades de forma greedy

```
Selección = ∅
Mientras queden actividades disponibles
        Obtener la actividad disponible "a" que finalice antes
        Selección = Selección ∪ {a}
        Desechar "a" y las actividades incompatibles con "a" de las disponibles
```

Inicialmente el emprendedor puede optar por seleccionar entre todas las actividades. La primera elegida será (C,1,4) que es la que finaliza primero que todas. Vemos que (D,1,10) y (G,3,6) son incompatibles con "C". Por lo que se descartan. La siguiente que selecciona corresponde a (B,4,5) que no tiene ninguna incompatibilidad. Posteriormente selecciona (E,6,8) y eso hace que (H,7,15) y (A,5,10) se desechen. Por último selecciona (I,10,12) y por lo tanto no podrá hacer (F,11,15). En resumen aceptará 4 excursiones: B,C,E,I.



Analicemos la optimalidad del algoritmo propuesto. En primer lugar vemos que el resultado obtenido es compatible entre sí. Esto se debe a que por cada elección removemos los incompatibles de las actividades disponibles. Entonces a la hora de obtener una actividad para seleccionar está no se solapará con las anteriores seleccionadas. En segundo lugar vemos que estamos seleccionando siempre la primera actividad que

termina y por lo tanto al dividir en los subproblemas solo generamos uno nuevo con las actividades compatibles que comienzan luego de esta. No hay subproblema previo, dado que no hay disponibles que terminen antes que la primera que termina. Esto mismo ocurre para cada subproblema resultante.

Falta determinar si el que elegimos está en una solución óptima. Para eso supongamos que conocemos una solución óptima a la que llamamos OPT. Llamamos a las actividades de OPT como el conjunto $\{o_1, o_2, ..., o_k\}$. Luego vemos la solución S que obtiene el algoritmo greedy. Llamamos a las actividades de S como el conjunto $\{s_1, s_2, ..., s_j\}$. En ambas soluciones consideramos que las actividades se encuentran ordenadas por el tiempo de finalización de las mismas. Si o_1 es la misma que s_1 entonces listo. En el caso que sean diferentes sabemos que - por el criterio de elección - s_1 termina antes o al mismo tiempo que o_1. Consideremos entonces la solución OPT sin la actividad o_1. Sabemos que la actividad o_2 comienza después que la actividad s_1, por lo que la actividad o_1 puede ser reemplazada por s_1 en OPT y el resultado aún será óptimo

Inductivamente podemos trabajar con los subproblemas resultantes. Continuamos reemplazando cada actividad i del óptimo por la actividad i de la solución S y la misma seguirá siendo óptima. En el proceso final la solución óptima genérica se habrá transformado en la solución greedy si la cantidad de actividades entre ambas soluciones es la misma. Por este comportamiento podemos decir que la elección greedy se “mantiene por delante” que cualquier solución óptima posible.

¿Qué podemos decir que la cantidad de actividades de cada uno de las soluciones? En principio $k \geq j$. Sino OPT no sería un óptimo. ¿Puede ser $k<j$? No, nuevamente por el criterio de elección y por el análisis que realizamos antes que este. La solución retornada por el algoritmo greedy se mantiene siempre “por delante” de la solución OPT. Esto significa que siempre s_i termina al mismo momento o antes que o_i. Entonces como o_{i+1} es compatible con O_i (sino OPT no sería una solución válida), no se superponen y por lo tanto comienza luego que termina el anterior. Y como si termina antes que o_i, entonces al momento que el algoritmo greedy selecciona si la actividad o_{i+1} aún no comenzó. y por lo tanto está aún disponible para elegir. Por lo que no es posible que $k<j$.

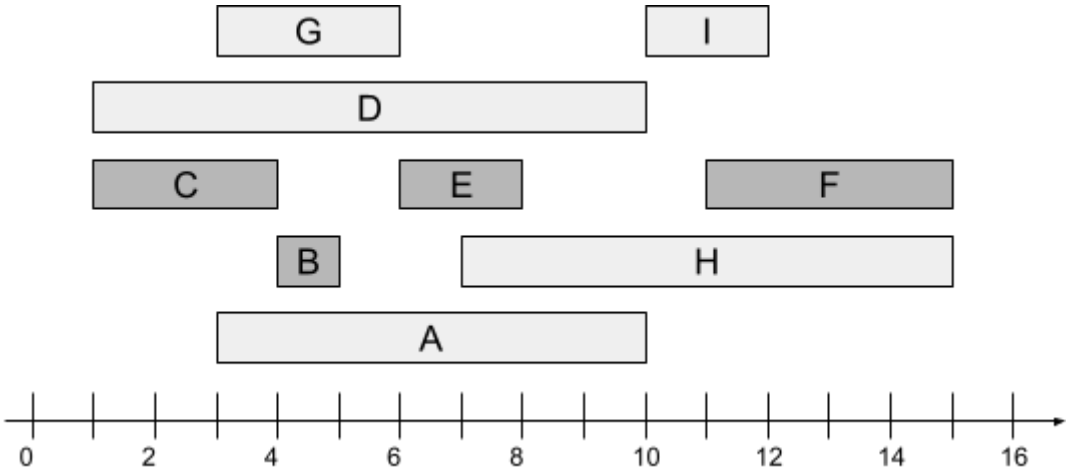
Uniendo todas las deducciones realizadas podemos afirmar que la solución greedy es una solución óptima.

Antes de finalizar con el análisis de optimalidad retornemos a los criterios de elección codicioso. Algunos de ellos nos quedaron por revisar. ¿Implica haber encontrado un criterio óptimo para desechar el resto? En algunos problemas puede ocurrir que existan más de un criterio de elección greedy válido. Este es uno de ellos. Analicemos el criterio de elección “el último que inicia”.

Cuando analizamos la subestructura óptima del problema vimos que dado una actividad seleccionada nos quedaban 2 subproblemas. Los compatibles que terminaban antes y los compatibles que comenzaban después. Con el criterio “el que finaliza primero” logramos que únicamente quede el subproblema de los que comienzan luego. Ahora, con el criterio “el último que inicia” podemos ver que sólo nos quedará un único subproblema con todos los compatibles que terminen antes. Podemos realizar el mismo proceso de demostración (intercambiando índices y criterios de compatibilidad) y demostrar que también es óptimo. Pero en este caso queda la rigurosidad para el lector. Realizaremos una explicación visual para entender por qué también es una elección greedy válida.

Podemos representar las actividades con sus tiempos de inicio y finalización en un gráfico temporal. Allí vemos que algunas actividades se solapan y por lo tanto son incompatibles entre sí. También podemos marcar aquellas seleccionadas por el algoritmo greedy u alguna otra solución óptima obtenida. En el eje x que comenzamos con el tiempo cero vemos que claramente todas las actividades inician antes que su finalización. Lo que vamos a hacer es rotar ese gráfico 180 grados. Ahora todas las actividades finalizan antes de iniciar en esta representación. Pero aun así una solución óptima sigue siéndolo en esa nueva visualización. Y vemos que el criterio del último que inicia se puede ver en esta representación como el que finaliza primero. Por lo tanto, iniciar desde el comienzo en la selección o desde el final son ambas estrategias totalmente válidas. Nos darán soluciones óptimas, aunque no necesariamente iguales.

Si el emprendedor selecciona teniendo en cuenta cual inicia más tarde vemos que las actividades seleccionadas finalmente son en algunos casos diferentes. Primero selecciona (F,11,15) descartando (I,10,12) y (H,7,15). Luego selecciona (E,6,8),haciendo incompatibles en el proceso a (A,5,10) y (D,1,10). Luego selecciona (B,4,5) y desecha (G,3,6). Por último incorpora (C,1,4). Las 4 actividades aceptadas serán F,E,B,C



Para una instancia particular del problema ambas elecciones greedy pueden dar el mismo o diferente subconjunto óptimo. Ambos son óptimos extremos. En el primer caso preferenciando liberar cuanto antes el recurso. En el segundo iniciando lo más tarde posible su ocupación. En el intermedio pueden existir muchas más variantes.

Tenemos ya un algoritmo greedy que es óptimo. Comencemos a analizar su eficiencia. Veremos que realizando elecciones incorrectas de implementación podemos terminar con un algoritmo no tán eficiente. En primer lugar buscar la próxima actividad si las actividades no tienen un orden determinado es un proceso que se realiza en $\mathbf{O}(n)$. Y en el peor de los casos se debe realizar $\mathbf{O}(n)$ veces (si ninguna actividad es incompatible entre sí. Una vez encontrada la próxima actividad a seleccionar se deben eliminar las incompatibles. Nuevamente nos encontramos en un proceso $\mathbf{O}(n)$. Por lo que estaremos frente a un algoritmo $\mathbf{O}(n^3)$. No es una buena noticia.

Por suerte lo podemos hacer bastante mejor. Si tenemos ordenados por el criterio de recorrido entonces podemos para obtener la próxima actividad recorrer el listado linealmente. Además podemos evitar recorrer continuamente para desechar las actividades incompatibles.

Se muestra a continuación el pseudocódigo con la propuesta. Se ve claramente allí la complejidad. Aunque en esta presentación el análisis de optimalidad no es tan evidente.

Seleccionar actividades de forma greedy (eficiente)

```
Ordenar las actividades de acuerdo al tiempo de finalización
Selección = {actividad[1]}
Fecha_fin = actividad[1].fecha_fin
Desde la actividad (i=) 2 a la n
        Si actividad[i].fecha_inicio >=Fecha_fin
                Selección = Selección  ∪ {actividad[i]}
                Fecha_fin = actividad[i].fecha_fin
Retornar Selección
```

En nuestro nuevo esquema tenemos un ordenamiento inicial de **O**(nlogn). Por otra parte recorremos esas actividades ordenadas linealmente en **O**(n). no hace falta en cada elección excluir de las actividades disponibles sus incompatibles. Esto es por que a medida que seleccionamos la fecha de finalización de la última tarea seleccionada irá creciendo. Por lo tanto aunque la actividad era incompatible con una anterior a la última seleccionada, como la fecha de inicio de la incompatible es menor a la de fin de la última seleccionada podemos descartarla indicando que es incompatible. Por lo tanto la complejidad temporal de la implementación es **O**(nlogn). El espacio requerido adicional es **O**(1) para almacenar la fecha de finalización de la última actividad seleccionada y en el peor de los casos si todas las actividades son seleccionadas requerimos **O**(n) para poder retornar las actividades seleccionadas.

4 Coloreo de intervalos

Nuestro próximo problema continúa en la senda de la planificación de ejecución de actividades. En este caso hay una diferencia fundamental. El recurso para ejecutar las actividades anteriormente estaba restringido en solo uno. Ahora contamos con un número limitado pero mayor a uno de recursos similares. Tenemos un conjunto de actividades con fechas de inicio y finalización. Algunas de esas actividades pueden ser incompatibles entre sí. Pero en vez de desechar unas de ellas, utilizaremos un recurso diferente para su ejecución. En un caso extremo podríamos asignar un recurso por cada actividad si nos alcanzan los recursos. Lo que transforma este problema en uno trivial. Aunque generalmente contamos con menos recursos que actividades. El objetivo a lograr es determinar si nos alcanzan los recursos y posiblemente minimizar la utilización de estos.

Se conoce a este escenario como el problema del coloreo de intervalos (interval coloring problem). La idea de asignar un “color” proviene de otros problemas donde el objetivo es pintar (marcar con diferentes identificadores) elementos para que no quede ninguno agrupado con otro del mismo color. El más conocido de ellos probablemente es el problema del coloreo de mapas. Otro nombre que recibe es problema de partición de intervalos (interval partitioning problem). Lo definiremos formalmente a continuación.

Coloreo de intervalos
Contamos con un conjunto de “n” actividades. Cada actividad x tiene una fecha de inicio i_x y una fecha de finalización f_x. Además contamos con un número “r” de recursos donde se pueden llevar a cabo estas actividades. Determinar si es posible cumplir con todas las actividades utilizando la menor cantidad de recursos posibles. Además, dar una programación posible para llevarlas a cabo.

Debido a motivos imprevistos toda la actividad de una jornada de actualización pediátrica se debe realizar en forma remota. Para lo cual debe contratar un conjunto de cuentas de streaming de videos para poder generar paneles online. Los organizadores tienen el detalle del horario de cada charla. Estas son (identificador, inicio,fin): (A,8:20,9:10), (B,8:50,9:30), (C,10:00,10:45), (D,13:00,14:00), (E,9:20,10:45), (F,14:20,15:50), (G,10:30,11:10), (H,16:20,17:30), (I,12:30,13:10). Un proveedor les ofrece un pack gratuito de 3 cuentas. ¿Pueden organizar las charlas de modo que esta cantidad les alcance?

Sabemos que si tenemos dos actividades que son incompatibles entre sí, entonces requerimos diferentes diferentes recursos para llevarlas a cabo. En un análisis similar podemos ver que si tres actividades coinciden en un punto en el tiempo serán requeridos 3 recursos. Generalizando, podemos afirmar que la cantidad máxima de recursos necesarios para una instancia en particular del problema corresponde a “k” la cantidad máxima de actividades que coinciden en un instante de tiempo determinado. Podemos llamar a esta característica de la instancia como “máxima superposición”. Si encontramos un algoritmo que utiliza tantos recursos como la máxima superposición, entonces podemos afirmar que el mismo es óptimo.

Una idea que suele surgir para enfrentar este problema es buscar primero el valor de máxima superposición. Luego intentar acomodar en ese número de recursos las actividades. En esa búsqueda suele aparecer el criterio de discretizar el tiempo en regiones y contabilizar en ellas cuantas actividades coinciden en estas. Generalmente - salvo en casos muy puntuales - esta idea puede llevarnos a resultados muy ineficientes. El algoritmo pasa a basar su complejidad en un parámetro que no depende del número de intervalos, sino de cómo estos se distribuyen en el tiempo. La cantidad de espacio extra requerido y el tiempo de ejecución estarán atados a esto. Llamamos a un algoritmo **pseudo polinomial** cuando la complejidad del mismo se mide en función de un valor numérico de la entrada y no la cantidad de elementos de esta. Seguir este camino nos enfrenta a este tipo de complejidad. Si la cantidad de regiones es mayor a la cantidad de actividades termina siendo más eficiente comparar cada actividad con el resto para obtener el valor buscado en $\Theta(n^2)$. Para sepultar la idea de proseguir por este camino queda indicar que una mala elección de las regiones puede llevar a suponer que dos actividades se superponen cuando la realidad es lo opuesto.

Si regionalizamos el tiempo para las diferentes charlas pediátricas debemos tener en cuenta que la primera de ellas comienza a las 8:20 hs y la última finaliza a las 17:30 hs. Hay un total de (9:10 hs a cubrir). Para evitar que dos actividades que compartan región aún pueden ser compatibles entre sí, se debe definir una duración de la región de 0:05 hs. Eso contabiliza un total de 110 regiones. Un valor bastante mayor a las 9 charlas a programar. Si bien esto se puede mejorar, parece más una complicación que una solución elegante.

Por suerte, como veremos a continuación, podemos construir un algoritmo de tipo greedy que solucione el problema de forma óptima. No se basa en obtener la máxima superposición. Sin embargo culmina encontrando una asignación de las actividades en la misma cantidad de recursos que este valor. Requiere encontrar una manera de ir procesando iterativamente cada una de las actividades y determinando un recurso a donde asignarlo. Nuevamente nos encontraremos con diferentes criterios para el orden de procesamiento. También diferentes criterios para establecer a qué recurso asignarlo.

La idea general será iniciar sin recursos utilizados. Tomar la próxima actividades según un orden establecido y asignarla siempre que sea posible a un recurso previamente utilizado.

Si no es posible, se debe agregar un recurso con esta actividad. Se repetirá hasta que todas las actividades estén asignadas o hasta que no queden recursos disponibles.

Analizaremos como opción el comenzar a asignar a las actividades que comienzan antes. Detrás de esta idea está el principio de comenzar a utilizar un nuevo recurso cuanto antes. No importa si la actividad dura poco o mucho, dado que igual la debemos ejecutar y durante ese tiempo cualquier otra actividad incompatible debería si o si usar otro recurso. Ubicamos a la actividad seleccionada en el primer recurso que encontremos disponible. El siguiente sería el pseudocódigo.

Colorear actividades

```
Ordenar las actividades por fecha de inicio
Enumerar los recursos de 1 a r.
Por cada actividad "a"
        Por cada actividad "prev" que precede y se superpone con "a"
                Excluir el recurso donde está "prev" para "a"
        Si existe un recurso que no fue excluido
                asignar al recurso no excluido de menor índice a "a"
        Sino
                Retornar "no hay suficientes recursos"
Retornar las asignaciones.
```

Las charlas ordenadas por fecha de inicio son (A,8:20,9:10), (B,8:50,9:30), (E,9:20,10:45), (C,10:00,10:45), (G,10:30,11:10), (I,12:30,13:10), (D,13:00,14:00), (F,14:20,15:50), (H,16:20,17:30). La charla A no tiene recursos a excluir. Asignamos "A" a la primera cuenta "c1". La charla "B" se superpone con "A" y por lo tanto se excluye "c1". Le queda la próxima cuenta "c2". Proseguimos con "E" que se superpone con "B" (en c2). Por lo tanto se asigna a "c1" (la menor disponible). La charla "C" se superpone con "E" (en c1) y por lo tanto queda asignada en "c2". Corresponde asignar la charla "G" que se superpone con "E" y "C". Quedará asignada en la cuenta "c3". Continuando el procesamiento, la charla "I" se asigna a "c1" y "D" a "c2" Finalmente las charlas "F" y "H" se asignan a la cuenta "c1". La cantidad de cuentas disponibles es adecuada para realizar todas las charlas y la asignación queda establecida.

¿Consigue un resultado óptimo este planteo? En primer lugar vemos que a medida que analizamos una actividad la vamos asignando a un recurso. Podemos ver esa asignación como el coloreo. La actividad tendrá un color (o etiqueta). El mismo que otras actividades

asignadas al mismo recurso. Podemos llamar a los colores genéricamente $\{c_1, c_2, c_3 \dots , c_r\}$. El algoritmo funciona en una pasada y teniendo únicamente en consideración las elecciones anteriores define el color de la actividad. Por esto podemos calificar a este como greedy.

Es clave el ordenamiento previo. Veamos con un ejemplo el resultado de no seguir una elección adecuada. Supongamos que sólo procesamos las actividades según arriban.

Tenemos que coordinar 4 charlas: (A,9:00,11:00) (B,14:00,15:00), (C,10:00,13:00), (D,12:00,16:00). Contamos con 2 cuentas para ingresarlas. Se prescindirá del ordenamiento por fecha de inicio. Tomamos la charla "A" y la asignamos a la cuenta "c1". Tomamos la charla "B" y como es compatible con "c1" la ingresamos en la misma. Es momento de la charla "C" que es incompatible con la charla "A" y "B" no podemos ponerla en la cuenta "c1". utilizamos para "C" la cuenta "c2". Finalmente es turno de la charla "D" que se superpone con "B" y "C". Por ese motivo no podemos ponerlas en ninguna de las dos cuentas. Siendo necesaria una tercera con la que no contamos. Sin embargo vemos que podríamos realizar una asignación diferente que solo requiere 2 cuentas. Asignamos "A" y "D" en la cuenta "c1" y "C" y "B" en la cuenta "c2". De igual manera si ordenamos por fecha de finalización y asignamos a la menor sala disponible en ese momento (o a la que se libera antes) llegaremos en el ejemplo a una solución no óptima.

En este algoritmo fallido tenemos actividades que se pueden superponer a la analizada tanto antes como después en la lista de actividades. Eso mismo sucede con nuestro algoritmo propuesto. Pero con una diferencia que será fundamental. Al ordenar por fecha de inicio sabemos que todas las actividades que inician antes incompatibles ya habrán sido asignadas. Allí radica el éxito del método.

El algoritmo en base a las incompatibilidades que inician antes descarta los "colores" en los que no puede estar. Deja aquellos en los que aún puede ser asignado. Pondrá la actividad en el color con el menor c_i $(1 \leq i \leq r)$ posible. ¿Por qué el menor posible? simplemente para no agregar recursos a menos que sea estrictamente necesario.

Sabemos que no puede haber más incompatibilidades que la máxima superposición "k". Supongamos que al momento de asignar una actividad encontramos que debe asignarse en el recurso k+1. Eso significa que "k" actividades que comenzaron antes son incompatibles con esta. Pero eso contradice el concepto de máxima superposición que

sería k+1. Como esto es absurdo implica que la manera de procesar y asignar las actividades nos asegura no superar el valor de “k” recursos.

Además, si al asignar una actividad ya asignó todos los colores posibles, entonces significa “K” es mayor a la cantidad de recursos “r”. Ergo, los recursos resultan insuficientes. Pero si K≥r entonces siempre se podrá asignar la actividad y el algoritmo finalizará con una asignación óptima de recursos.

El algoritmo es óptimo. ¿Pero qué tan eficiente es? Por un lado tenemos un ordenamiento lo que nos da una complejidad de $\mathbf{O}$(n log n). Luego tenemos que procesar cada uno de los recursos lo que agrega un $\mathbf{O}$(n). En ese procesamiento tenemos que revisar todas las actividades previas - nuevamente un proceso $\mathbf{O}$(n) - para encontrar a las incompatibilidades. Tenemos un tiempo $\mathbf{O}$(r) para la búsqueda del recurso disponible . Finalmente la asignación de la actividad en el recurso en $\mathbf{O}$(1). Esos tiempos lo podemos expresar como $\mathbf{O}$(n log n + n(n+r+1)). Simplificado en $\mathbf{O}$(n^2). En cuanto al espació requerimos únicamente $\mathbf{O}$(n) para almacenar las asignaciones a los recursos y $\mathbf{O}$(r) para ir determinando qué recurso está disponible para la asignación de un recurso. En conjunto requerimos $\mathbf{O}$(n+r) de espacio adicional.

Si bien podemos afirmar que es un algoritmo bueno (según la definición formal) veremos que podemos hacerlo mejor. No podemos evitar el ordenamiento. Pero si podemos mejorar la búsqueda de un recurso donde asignarlo. Observar lo que se busca es un recurso disponible y para eso se analizan todos las actividades pasadas. Esto se puede evitar simplemente manteniendo por cada recurso la fecha de liberación del mismo. Si esta es posterior a la fecha de inicio de la actividad implica que en ese recurso se encuentra programada una actividad incompatible con ella. Agregando entonces este valor podemos lograr que la complejidad total sea $\mathbf{O}$(n log n + n(r+1)). Simplificado en $\mathbf{O}$(n log n + nr).

Pero aún lo podemos hacer mejor. Al momento de programar una actividad obtenemos aquel recurso utilizado que se libera más tarde. Si la fecha en la que este se libera es posterior a la fecha de inicio de la actividad, entonces la actividad es incompatible con todos las actividades que se están ejecutando en todos los recursos. Se debe agregar en otro recurso. Si, por el contrario, este recurso puede ejecutar esta actividad, la asignaremos ahí. Para lograr esta idea podemos utilizar una cola de prioridades máxima para guardar los recursos donde la clave será la fecha de finalización de la última actividad

asignada a este. Si usamos un heap binario para la implementación de la cola lograremos que las operaciones para la gestión de la cola se **O**(log r). Quedando finalmente como complejidad temporal de nuestro algoritmo **O**(n (log n + log r)) o **O**(n (log nr)). Se muestra en el pseudocódigo la implementación de esta última idea.

Colorear actividades greedy (más eficiente temporalmente)

```
Ordenar las actividades por fecha de inicio
Sea H cola de prioridad máxima
Sea k = 0
Por cada actividad act
        obtener recurso "rec" de H con máxima clave
        Si act.fechaInicio >  rec.fechaLiberacion
                asignar act a recurso rec
                agregar (rec, act.fechaFinalizacion) a H
        Sino
                k++
                Si k<=r
                        obtener un nuevo recurso rnew
                        asignar act a recurso rnew
                        agregar (rnew, act.fechaFinalizacion) a H
                Sino
                        Retornar "no hay suficientes recursos"
Retornar las asignaciones.
```

En cuanto a la complejidad espacial el algoritmo requiere **O**(r) para la cola de prioridad y **O**(n) para guardar la asignación de cada recurso. Quedando finalmente en un **O**(n+r).

Para concluir podemos hacer un breve análisis del tipo de solución óptima conseguida. Se puede ver que muchas veces existirán diferentes asignaciones (o coloreo) para una instancia que resultaron óptimas. Si se analizan los dos algoritmos propuestos greedy una misma instancia puede darnos diferentes soluciones. Esto se debe a la forma de se prioriza la asignación de los recursos. En el primero aquel disponible de índice menor y en el segundo aquel que se disponibiliza más tarde. Existen otros algoritmos que también resuelven en forma eficiente. Por ejemplo utilizando para ordenar la fecha de terminación en forma decreciente y asignando al recurso que comienza antes que sea compatible (la podemos ver como una solución espejada a la antes descripta).

5 Camino mínimo

El uso de grafos para representar diferentes situaciones es una herramienta de probada utilidad. Un grafo corresponde a relaciones binarias entre conceptos. Representamos a estos como nodos y a su relación como ejes que los unen. Es común también llamar a los nodos como vértices y los ejes como aristas. Estos ejes pueden ser dirigidos (que parten de un nodo y finalizan en otro) o no dirigidos (el camino se puede realizar indistintamente entre ambos nodos). Además estos ejes pueden tener un valor numérico asociado. El uso de grafos se puede utilizar para modelar relaciones en redes sociales, una red de transporte que conecta diferentes ciudades, tareas dentro de un proyecto de desarrollo y muchas otras situaciones. Dentro de estas en ocasiones surge la necesidad de determinar el mejor camino (de menor costo) que une un determinado nodo con otro. En esta sección analizaremos justamente este problema. Pondremos algunas restricciones y condiciones. En principio el grafo puede ser tanto dirigido como no dirigido. Pediremos que exista un costo en cada eje y que el mismo sea un valor numérico positivo. Enunciamos formalmente el problema.

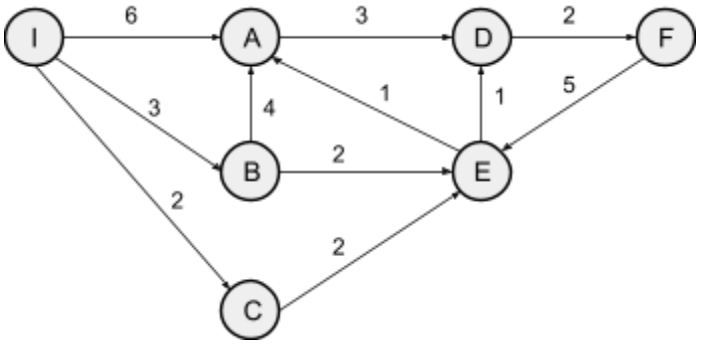
Camino mínimo en un grafo
Dado un grafo G=(V,E) de n vértices y m ejes con pesos positivos. Partiendo de un vértice x ∈ V deseamos conocer el camino mínimo al resto de los vértices de G

En el análisis del problema podremos encontrar definiciones y propiedades que nos ayuden a resolverlo en forma eficiente. Se denomina camino i-f a un recorrido que parte por un nodo inicial “i” y utilizando ejes del grafo va pasando por nodos intermedios hasta llegar a un nodo final “f”. Podemos describirlo como la sucesión de nodos por las que pasa n_0, n_1, n_2, ..., n_l donde n_0 corresponde al eje inicial “i” y n_l al nodo final “f”.

Llamaremos la longitud del camino a la cantidad de ejes por los que pasa. Un camino de longitud 1 no tiene nodos intermedios. Un camino de longitud “l” pasa por “l+1” nodos y (si l>2) contiene “l-1” nodos intermedios. Llamaremos ci al costo del eje i. El costo de un camino será la suma de los costos de los ejes por lo que pasa. En un grafo pueden existir diferentes caminos que unan a dos nodos (o ninguno). Queremos encontrar aquel de

menor valor entre todos ellos. Si nos encontramos en un grafo no conexo o no fuertemente conexo (en el caso que sea dirigido) existirán nodos en los que no existe un camino que los una. En ese caso diremos que tanto el costo como la longitud entre esos nodos es ∞.

Utilizaremos el grafo G, correspondiente a la imágen, con 7 nodos y 10 ejes. Partiremos del nodo "I" y queremos obtener el camino mínimo desde este al resto de los nodos. El costo de cada eje de muestra sobre el mismo. Se puede observar que para algunos nodos existen varios



caminos alternativos. Por ejemplo para llegar al nodo A se puede optar por los caminos 1: I,A 2: I,B,A 3: I,B,E,A y 4: I,C,E,A. Respectivamente los costos de estos caminos son 6, 7, 6 y 5. Por lo tanto este último corresponde al camino mínimo.

El algoritmo para solucionar este problema corresponde al propuesto por Edsger Dijkstra en 1956. Según sus propias palabras en una entrevista que concedió en el 2001[1] fue una idea que se le ocurrió tomando un café en 20 minutos en compañía de su prometida. Su idea era encontrar un problema sencillo que se pueda utilizar para explicar al público general no especializado sobre las bondades de una nueva computadora que iba a presentar. Recién tres años después en 1959 publicó su propuesta en el artículo "A Note on Two Problems in Connexion with Graphs" [2]. En su versión inicial solo calculaba la distancia mínima entre dos nodos. Sin embargo con mínimas modificaciones se puede utilizar para encontrar el camino mínimo entre un nodo de partida y el resto de los nodos del grafo.

El algoritmo de Dijkstra construye progresivamente las distancias mínimas. Para el procesamiento se dividen los nodos del grafo en tres conjuntos: alcanzados, frontera y no alcanzados. Los nodos alcanzados corresponden a aquellos a los que ya se conoce su

[1] Frana, Phil, An Interview with Edsger W. Dijkstra, Agosto 2010, Communications of the ACM.
[2] Dijkstra, E. W., A note on two problems in connexion with graphs, 1959, Numerische Mathematik.

distancia mínima. Entre los nodos alcanzados podremos ver que existen ejes que salen de ellos. Algunos incidentes en nodos del mismo conjunto y otros no. Al conjunto de todos los nodos no pertenecientes al conjunto de alcanzados a los que inciden ejes de este último los llamaremos frontera. Los nodos no alcanzados serán el resto.

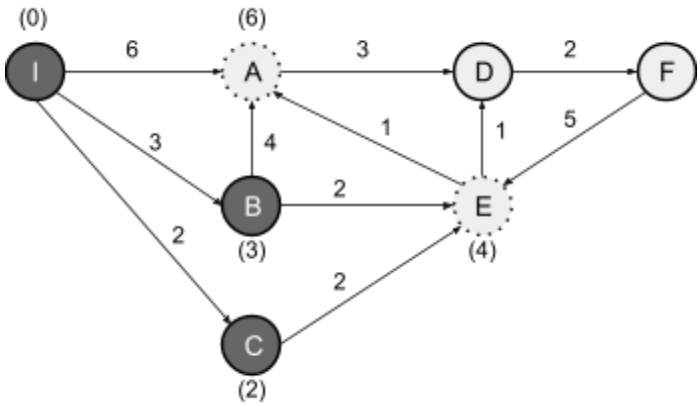
Al comenzar considera que el único nodo alcanzado es el origen “i”. Su distancia mínima a sí mismo es cero (costo cero). Los nodos alcanzables desde “i” (para los que existe un eje que parte de “i” y arriba a estos) tendrán un costo potencial de llegada igual al costo del eje que los une más el costo de llegar al nodo del que parte (en este caso cero). Estos pertenecen al conjunto frontera. El resto de los nodos por lo tanto serán los no alcanzados y tendrán un costo potencial de llegada infinito. En cada iteración del algoritmo se selecciona el nodo “s” del conjunto frontera con menor costo potencial de llegada. Este ahora formará parte del conjunto de alcanzados y su costo ya no será potencial sino el costo final y mínimo. Al modificarse el conjunto de alcanzados se producirán cambios en el resto de los conjuntos. Aquellos nodos no alcanzados que son accesibles desde el nodo seleccionado pasarán a formar parte de la frontera. Su costo potencial será el costo de llegar a “s” más el costo del eje que parte de “s” y llega a ellos. Por otra parte pueden aparecer nuevos caminos que unen nodos previamente incluidos en la frontera que pasen por el nodo “s”. Estos podrían ser menores que los previamente descubiertos. Por lo tanto se debe verificar cada uno de los nodos “x” en la frontera accesibles desde “s” y determinar si es el caso. Si el costo potencial previamente conocido a “x” es mayor al costo de acceder a “s” y de allí pasar a “x” entonces el nuevo camino descubierto es preferible. Por lo que este se actualizará para reflejar esta realidad. Este proceso se repite hasta que no quede ningún nodo en la frontera que equivale a que todos los nodos accesibles desde el nodo inicial están en el conjunto de alcanzados.

A continuación se describe mediante pseudocódigo la solución. Se agrega el almacenamiento del predecesor en el camino mínimo de cada nodo. De esta forma se podrán reconstruir los caminos obtenidos.

Camino mínimo - Dijkstra
frontera = ∅ costo[i]=0 y predecesor[i]=i Alcanzados = {(i,costo[i],predecesor[i])}

```
costo[x]=∞ para todo nodo x diferente a i
Por cada nodo "s" accesible desde "i"
        Agregar a frontera "s" con costo potencial costo[s] =C[i,s]+costo[i] y predecesor[s]=i
Mientras la frontera ≠ ∅
        Sea "s" nodo en frontera con menor costo costo[s]
        Quitar s de frontera
        Alcanzados = Alcanzados ∪ {s}
        Por cada x accesible desde "s" y x ∉ Alcanzados
                nuevoCosto = costo[s] + C[s,x]
                Si x ∈ frontera y nuevoCosto < costo[x]
                        predecesor[x] = s
                        actualizar en frontera x con costo[x]=nuevoCosto
                Si x ∉ frontera
                        predecesor[x] = s
                        agregar en frontera x con costo[x]=nuevoCosto
Retornar Alcanzados, Costo y predecesor
```

En el grafo presentado anteriormente, iniciamos con un único nodo alcanzado, el nodo I con costo de 0. Como los nodos A,B y C son accesibles desde I estos conformarán la frontera inicial. El resto serán no alcanzados. Rotulamos a los no alcanzados con costo infinito. Los costos potenciales a los nodos de la frontera serán costo[A]=6, costo[B]=3, costo[C]=2. En la primera iteración se selecciona al nodo C que corresponde al nodo de la frontera con menor costo potencial. El nodo C pasa a pertenecer al conjunto de los alcanzados. El nodo E se agrega a la frontera con costo potencial de costo[E]=4 (el costo de acceder a C más el costo del eje de C a E). No hay otros nodos que se actualicen. En la segunda iteración se selecciona el nodo B que ahora formará parte del conjunto de los alcanzados con costo 3. No hay nodos nuevos en la frontera. Pero desde B se puede acceder también a A y E. Esto nos da nuevos caminos posibles a estos. En ninguno de los casos los nuevos costos serán menores a los existentes (los nuevos costos serían 7 y 5). Por lo que ninguno de los costos se actualizan. En la imagen se muestra el estado del procesamiento luego de esta iteración. Tenemos a los nodos I,B,C como alcanzados. A y E como frontera. D y F como no alcanzados.



En la tercera iteración se selecciona el nodo E que pasará a estar alcanzado con costo 4. Se agrega a la frontera al nodo D con costo[D]=5. Además en este caso es conveniente acceder al nodo A que ya se encuentra en la frontera desde E con con un costo[A]=5 resultante. En las últimas tres iteraciones se seleccionará A con costo 5, D con costo 5 y F con costo 7.

Llegado a este punto es más que esperable preguntarse por la optimalidad del método. Si bien hablamos de un algoritmo de más de 60 años y muy conocido nosotros no podemos dejar de demostrarlo. Vemos que claramente corresponde a un algoritmo greedy. Es un algoritmo iterativo que progresivamente va agregando nuevos nodos generando nuevos subproblemas. En cada uno de ellos elige de forma codiciosa: siempre el nuevo nodo seleccionado será aquel que puede accederse con el menor costo entre todos los disponibles.

Al terminar la ejecución se habrá construido un árbol cuya raíz corresponde al nodo inicial y contendrá el camino mínimo desde la raíz a aquellos nodos con los que está conectado. Sabemos que es un árbol puesto que una vez conectado un nodo en la iteración no vuelve a tenerse en cuenta para futuras uniones desde otros nodos (se elimina de la frontera). Nos falta demostrar que esa unión corresponde al camino mínimo. Lo realizaremos demostrándolo por inducción.

Al iniciar únicamente está alcanzado el nodo inicial con costo cero. Desde el mismo se pueden acceder a un subconjunto de nodos en la frontera. Son aquellos que tienen un eje directo que parte del nodo inicial “I” y llega a ellos. Cada uno cuenta con un costo asociado. La elección greedy determina seleccionar aquel de menor valor. A partir de ese momento el nodo al extremo de ese nodo “X” pasa a ser alcanzado con el costo correspondiente a

ese eje más cero. ¿Puede existir un camino alternativo a X de menor costo? El resto de los caminos sólo pueden pasar por alguno de los otros nodos en la frontera. Todos ellos con un costo de eje mayor (en caso extremo igual). La longitud de ese camino alternativo tendrá al menos un eje más. Este tiene que tener un costo positivo (o en caso extremo cero) por las condiciones del enunciado del problema. Por lo tanto el encontrado corresponde al camino mínimo. Podrá existir otro de igual costo, pero nunca de menor. Luego de esa elección se actualiza la frontera agregando nuevos nodos y actualizándose los costos de forma que siempre se mantengan los costos mínimos de los ejes desde los alcanzados a la frontera.

Al comenzar la iteración n, tendremos n nodos seleccionados y el algoritmo seleccionará al nodo n+1. La elección será entre aquellos que se encuentren en la frontera. El costo de cada uno de ellos corresponde al costo de llegar desde el nodo de inicio a alguno de los nodos alcanzados más el costo del eje para alcanzarlo. Pueden existir varios caminos pero siempre nos quedamos con el menor de ellos. Tenemos que ver dos cosas diferentes. Si el costo de elegir ese nodo en este momento podría ser mayor a algún punto pasado de la ejecución del algoritmo. O si el costo de alcanzar a este nodo podría ser menor en un futuro paso del algoritmo.

La primera parte es sencilla de responder. A partir de la inclusión del nodo en la frontera por condición del algoritmo el costo solo puede mantenerse o disminuir. El nodo se incluye en la frontera cuando se conoce al menos un camino desde el nodo inicial. Por lo tanto esperar para su inclusión sólo puede ser beneficioso para el mismo. ¿Pero cuánto esperar? La respuesta corresponde a la segunda cuestión. Su inclusión tiene que asegurarnos que no hay forma que en el futuro exista un camino mejor.

Todos los nodos alcanzados corresponden a un árbol cuya raíz es el nodo inicial. Podemos incluir como nodos terminales de ese árbol a los nodos frontera. Se unirá al nodo alcanzado con el que logre el menor costo (como bien afirmamos antes). De todos los nodos fronteras seleccionamos aquel con el menor costo de llegada y lo convertimos a un nodo alcanzado. Es imposible que más adelante encontremos un camino alternativo a este con menor costo. Siguiendo el árbol recién armado actualizado este nuevo camino debería empezar por en nodo inicial y antes de finalizar pasar por alguno de los otros nodos terminales que todos ellos tienen un costo mayor al seleccionado. Como todos los ejes

tienen costo positivo (o en el caso extremo cero) no es posible otro camino de menor costo (en el caso extremo uno de igual costo).

Por todo lo antes enunciado concluimos que el algoritmo de Dijkstra es óptimo para cualquier instancia del problema.

La complejidad espacial y temporal dependerá de las estructuras seleccionadas para la implementación. Una solución eficiente requiere poder obtener para un nodo determinado cuales son sus ejes salientes. Para eso podemos representar el grafo utilizando una lista de adyacencias. El espacio requerido para esta estructura $\mathbf{O}(n+m)$. Los nodos alcanzados los podemos ir registrando en un vector para su posterior devolución con su correspondiente costo y nodo predecesor. Requiere como espacio $\mathbf{O}(n)$. Para almacenar los predecesores de cada nodo podemos utilizar un vector con requerimiento de espacio de $\mathbf{O}(n)$. Podemos utilizar una cola de prioridad mínima con soporte de actualización de claves para almacenar los nodos en la frontera. La clave consistirá en el costo de acceder al nodo en cuestión. Tendremos como máximo n-1 elementos en la cola de prioridad (si todos los nodos son accesibles desde el nodo inicial). Por lo tanto requerirá un espacio de $\mathbf{O}(n)$. En conjunto el algoritmo presenta una complejidad espacial de $\mathbf{O}(m+n)$.

Con respecto a la complejidad temporal, podemos ver que tendremos como máximo n iteraciones (puesto que en cada iteración seleccionamos un nodo como alcanzado). En cada iteración obtenemos el nodo "s" con menor costo de la cola de prioridad en $\mathbf{O}(f_{obt\_min})$. Para el elemento lo agregamos en la lista de alcanzados en $\mathbf{O}(1)$. Luego en la lista de adyacencias recorremos los nodos accesibles desde "s". Llamaremos a este valor "E_s". Por cada uno de estos nodos veremos si previamente está alcanzado, si se debe agregar a la frontera o si se debe actualizar su costo por uno menor en la misma. Respectivamente se puede hacer en $\mathbf{O}(1)$, $\mathbf{O}(f_{insertar})$ y $\mathbf{O}(f_{dec\_clave})$. El costo de operación f_{obt_min} , $f_{insertar}$ y f_{dec_clave} dependen del tipo de cola de prioridad utilizada (por ejemplo: heap binario, binomial o fibonacci). El costo temporal de actualizar el predecesor y de actualizar el costo es $\mathbf{O}(1)$.

Con todas las partes dispuestas, encontrar la expresión general que representa la complejidad temporal del algoritmo puede resultar algo complejo. Pero podemos ir encontrando ciertos atajos que nos ayuden. Algo útil a tener en cuenta es que si el grafo es conexo (o fuertemente conexo si es dirigido) entonces eventualmente todos los nodos serán alcanzados. Tendremos un total de n f_{obt_min} y n agregar a alcanzados de $\mathbf{O}(1)$. Además

en cada iteración por cada nodo alcanzado recorreremos todos sus sucesores, que corresponde al número de ejes que abandonan al nodo. Podemos notar que $\sum_{n \in V} E_n = m$ por lo que el proceso "Por cada x accesible desde "s"" se realizará un total de m veces correspondiente al número total de ejes (si el grafo es no dirigido corresponde a 2m). En estos se verifica si el nodo está previamente alcanzado en **O**(1). Luego, el $f_{insertar}$ se puede invocar como máximo n veces y el f_{dec_clave} como mucho m-n veces (Lo podemos simplificar a m si la diferencia entre ellos es suficientemente grande).

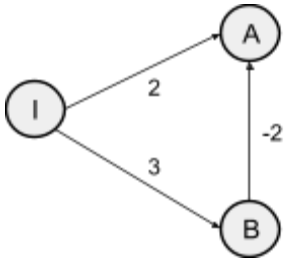
Uniendo todas las partes podemos decir que la complejidad es **O**(n*(cte+ f_{obt_min} + $f_{insertar}$) + m*(cte+f_{dec_clave})). Simplificando **O**(n*(f_{obt_min} + $f_{insertar}$) + m*(f_{dec_clave})). La siguiente tabla muestra la complejidad resultante según el tipo de cola de prioridad utilizada:

Cola prioridad	f_{obt_min}	$f_{insertar}$	f_{dec_clave}	Complejidad
Heap Binario	**O**(logn)	**O**(logn)	**O**(logn)	**O**((n + m)*logn)
H. Binomial	**O**(logn)	**O**(1)	**O**(logn)	**O**((n + m)*logn)
H. Fibonacci	**O**(logn)	**O**(1)	**O**(1)	**O**(n*logn + m)

Nos resta analizar la solución encontrada. Como se afirmó anteriormente pueden existir diferentes caminos mínimos a un mismo nodo. En el algoritmo presentado al momento de actualizar el costo mínimo de los nodos en la frontera si el nuevo encontrado es igual al preexistente se está descartando. Cambiando esta condición aun se encontrará una solución óptima, potencialmente diferente. Pueden existir un número exponencial de soluciones posibles. Esto se puede ver si nos encontramos frente a un grafo completo donde todos los ejes tienen el mismo costo. Para obtener todas las soluciones se pueden almacenar en vez de únicamente un nodo predecesor, una lista con todos los posibles predecesores con el mismo costo de camino. Reconstruir todos los caminos en ese caso corresponderá a un problema de enumeración combinatorio y tendrá una complejidad no polinómica.

Antes de concluir es importante mencionar que si se relajan las condiciones del costo de los ejes y se permiten valores negativos este algoritmo no garantiza la optimalidad de la solución. Seleccionar el nodo de la frontera con el menor costo no garantiza el menor

camino al mismo. Dado que por otro camino tal vez se pueda llegar a un nodo con un valor negativo que lo mejore. Eso se puede comprobar mediante un simple ejemplo de 3 ejes que se muestra a continuación. El algoritmo de Dijkstra encontrará el camino con costo 2 I,A. Sin embargo se puede comprobar que el menor camino corresponde a I,B,A con un



costo de 1.

6 Árbol recubridor mínimo

Un problema de grafos muy conocido y de amplia utilidad corresponde a la obtención de un árbol recubridor mínimo (minimum spanning tree). De forma similar al problema del camino mínimo nos interesa que exista un camino que conecte a cada uno de los nodos del grafo. La diferencia radica en que no esperamos que ese camino sea necesariamente el mínimo entre todos los posibles del grafo. Queremos que la suma de los costos de los ejes seleccionados del grafo que permitan la interconexión global sean lo menor posible. Podemos encontrar utilidad en este problema dentro situaciones en la práctica como interconectar redes informáticas, generación de carreteras entre ciudades, redes de tendido eléctricos y muchas otras. Para este problema solicitaremos nuevamente que los pesos de los ejes sean positivos. Agregaremos la condición que sea un grafo no dirigido.

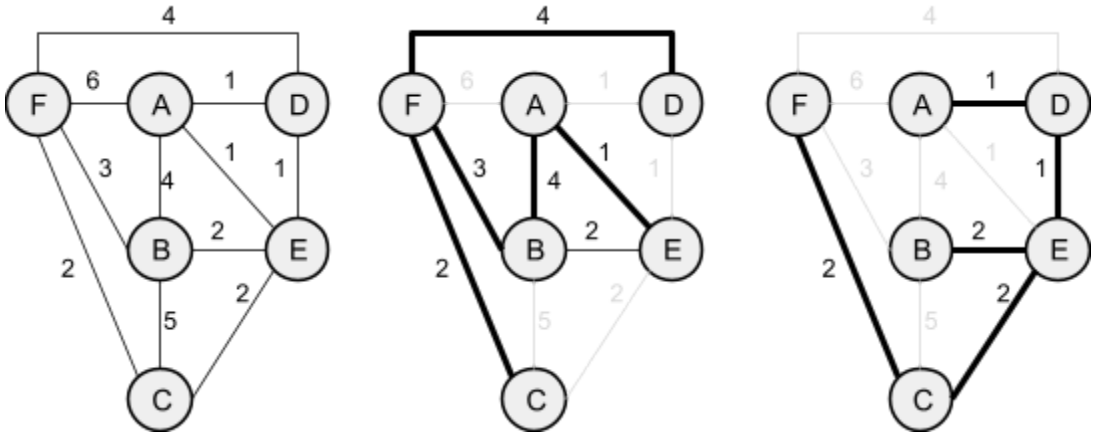
Formalmente lo definimos el problema como:

Árbol recubridor mínimo
Dado un grafo G=(V,E) conexo y no dirigido de n vértices y m ejes con pesos positivos. Seleccionar un subconjunto de ejes que asegure la conexión entre todos los nodos y que minimice la suma de los costos de ellos.

Para comenzar el análisis del problema primero nos enfocamos en un “spolier” que el mismo nombre del problema nos indica. La mínima cantidad de ejes que se puede

seleccionar para unir todos los “n” nodos dentro de un grafo es “n-1” ejes. Se puede comprobar que solamente agregando uno más de ellos se conformará un ciclo en el grafo y con eso un camino alternativo entre al menos 2 nodos en el grafo. Si analizamos ese ciclo vemos que removiendo solo 1 eje la conectividad del grafo no se romperá y que al ser todos los ejes de costo positivo se reducirá el costo total de la solución. En teoría de grafos la definición de un **árbol** corresponde a un grafo conexo acíclico y no direccionado (connected acyclic undirected graph). Si bien el enunciado del problema no habla de construir un árbol, el resultado óptimo de resolver el problema deberá serlo.

Antes de continuar veamos un breve ejemplo del problema



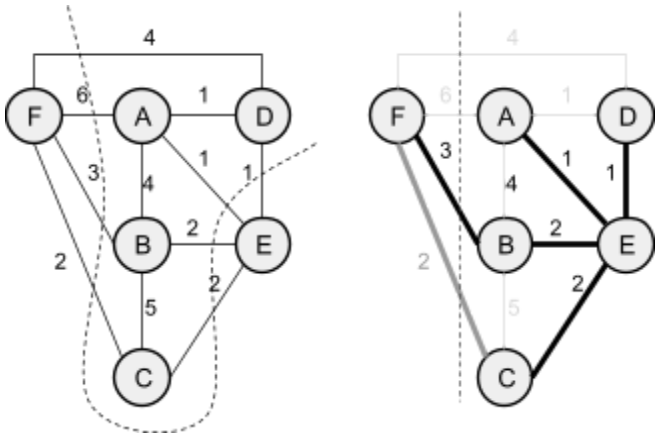
El primer grafo presentado corresponde a una instancia posible del problema. Tenemos 6 nodos y un conjunto de 11 ejes con su correspondiente peso. El segundo y tercer grafo corresponden a la misma instancia anterior con diferentes propuestas de solución. Se remarcan un subconjunto de ejes que aseguran la interconexión de todos los nodos. El agregado de cualquier otro eje es redundante, innecesario y agrega costo global. La suma de los costos de los ejes seleccionados es 14 y 8 respectivamente. Claramente el último es el preferible entre ellos.

Llamamos un **corte** (cut) en un grafo a una separación imaginaria de sus nodos en dos conjuntos disjuntos. Cada conjunto deberá tener al menos un nodo. Se puede observar que la cantidad de cortes posibles es 2^{n-1}. Para un set particular podemos encontrar un subconjunto de ejes que tienen sus extremos en nodos de diferentes subconjuntos de la partición. Llamaremos a ellos el conjunto corte (cut-set). En la solución óptima sabemos que al menos un eje del cut-set tiene que estar seleccionado en el árbol recubridor. De lo contrario no existirán caminos entre los nodos de los conjuntos de cada lado del corte.

Tenemos que tener en cuenta que pueden existir en la solución óptima varios ejes que cruzan de un lado al otro del corte. Por ejemplo, nada impide que de un mismo lado de un corte queden varios componentes inconexos entre sí.

La siguiente propiedad que estudiaremos será de suma importancia al momento de crear un algoritmo greedy para resolver el problema. Dado un corte en el grafo, el eje de menor costo que integra el conjunto de corte debe pertenecer al árbol recubridor mínimo. Probaremos esto por el absurdo. Supongamos que conocemos la solución óptima

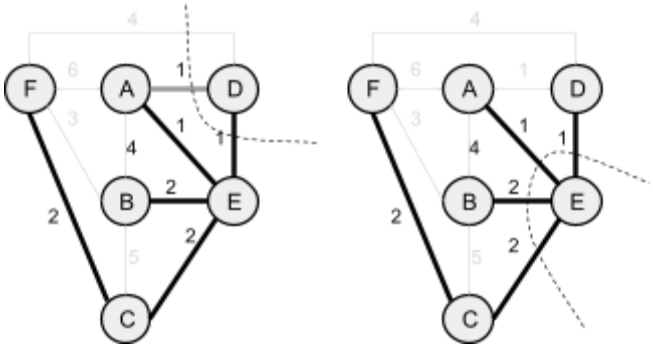
S de una instancia y que encontramos dentro de la misma un corte donde el eje de menor costo "x" no pertenece al conjunto de ejes de la solución. En este momento existirán n ejes para n nodos. Tendremos un ciclo que deberá contener el eje agregado y otro que pasa por el corte "y". Si removemos el eje "y" volveremos a tener un árbol recubridor. Pero con un costo menor al del S. Por lo tanto arribamos a una contradicción probando nuestra propiedad.



Se observan dos cortes en el grafo del ejemplo. En el primero de ellos se divide a los nodos en los conjuntos F,E y A,B,C,D. Como se puede observar no existen ejes que conectan a los nodos del primer conjunto entre sí. Por lo tanto en la solución óptima tendremos más de 1 eje que pase por ese corte. En el segundo corte se presenta además en supuesto árbol recubridor mínimo (remarcado en negro). En el corte presentado se ve que no se encuentra presente el eje de menor costo (F-C). Si se le agrega se produce un ciclo F,B,E,C,F. Los nodos F-B e F-C pertenecen al mismo y al conjunto de corte. Removiendo el eje F-B volvemos a un árbol cuyo costo se modifica (la diferencia entre el agregado y el removido) pasando de 9 a 8. Evidenciando que la sugerida no era una solución óptima.

Cuando en un grafo no existen dos o más ejes con el mismo costo, entonces existe una única solución óptima al problema. Esto ocurre porque por cada corte posible que realice siempre existirá un eje menor y ese por la propiedad planteada deberá pertenecer al árbol solución. Sin embargo, puede existir más de un eje con el mismo costo y tal vez sea posible construir un corte en el que formen parte del conjunto de corte y compartan la condición de ser el de menor costo. ¿Qué ocurre en ese caso? ¿Pertenecen ambos a la solución? ¿solo uno? ¿Ninguno?

Claramente podemos desechar que el árbol recubridor mínimo no incluya ninguno de ellos. Ya demostramos que el eje de menor en un conjunto de corte debe pertenecer a la solución óptima. Al menos uno de los dos debería estar. Llamemos a estos ejes x e y. Supongamos que tenemos una solución óptima con sólo “x”. Si agregamos “y” y se produce un ciclo que incluye a “x” entonces vemos que esos ejes son mutuamente excluyentes. Y podemos dejar a cualquiera de los dos. Esto nos muestra que pueden existir soluciones alternativas igualmente óptimas. También puede ocurrir que ambos pertenezcan a la solución. Eso ocurrirá si cada uno de ellos es el menor de los ejes en otro corte donde no esté su contraparte. En definitiva podemos elegir a uno de ellos de forma arbitraria y afirmar que pertenece a un árbol recubridor mínimo. La elección de uno sobre el otro determinará un subconjunto de soluciones óptimas de la instancia que podremos construir.



En el primer grafo se muestra un posible árbol recubridor mínimo con un corte donde dos ejes de costo 1 corresponde a los ejes de menor costo. Al incluir ambos se genera un ciclo. Podemos dejar cualquiera de ellos y la solución se mantiene óptima. En el segundo grafo se muestra el mismo grafo con un corte donde dos de los ejes de menor costo pertenecen a la solución

óptima. Si se presta atención en el subgrafo A,E,D veremos que tenemos 3 alternativas de soluciones óptimas modificando que 2 ejes incluir.

Llegados a este punto conocemos un poco más sobre la naturaleza del problema, pero aun no contamos con un algoritmo bueno que lo resuelva. Para nuestra fortuna varios investigadores a lo largo de la historia han encontrado algunos de ellos. Veremos algunas de estas propuestas que utilizan la metodología greedy de solución.

El **algoritmo de Prim** fue descrito varias veces en la historia. Se cree que quien primero lo presentó fue Vojtěch Jarník[3] en 1930 en idioma checo. Seguramente por el idioma y geografía algunas décadas más tarde fue "redescubierto". Cronológicamente primero por Robert Prim[4] en 1957 y luego por Edsger Dijkstra[5] en 1959. El crédito popular se lo ha llevado Prim puesto que el método es reconocido por su nombre.

Corresponde a un algoritmo greedy. Trabaja separando los nodos en dos conjuntos. Llamaremos al primero A y al segundo B. Inicialmente selecciona un nodo "x" de forma arbitraria y lo introduce al conjunto A. El resto de los nodos pertenece a B. Esta separación corresponde a un corte del grafo y el conjunto de corte estará conformado por los ejes que tienen un extremo en el nodo "x". El algoritmo selecciona el eje de menor valor entre ellos (o en caso de empate arbitrariamente uno) y establece que el mismo pertenece a la solución. El nodo al extremo del eje seleccionado se quita del conjunto B y se coloca en el conjunto A. Este cambio genera un nuevo conjunto de corte con los ejes que tienen extremos en nodos de diferente conjunto. Nuevamente se selecciona el de menor valor y se repite el proceso hasta que no queden nodos en el conjunto B o que en un determinado momento el conjunto de corte sea vacío (esto puede ocurrir antes de vaciar el conjunto B, si el grafo no es conexo)

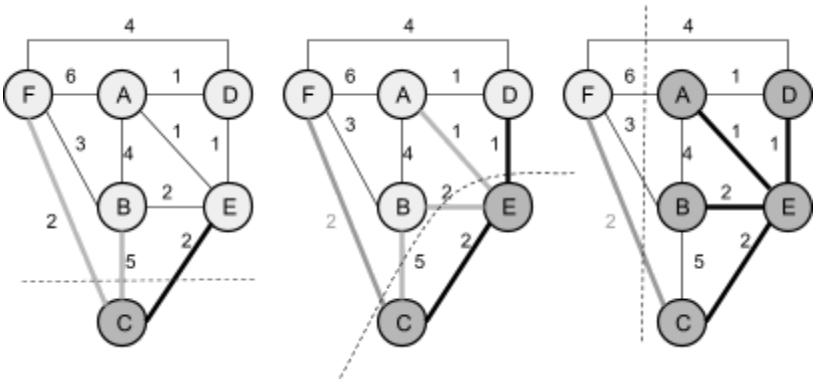
Árbol recubridor mínimo - Prim

```
A = x
B = V - {x}
arbol = ∅
```

[3] Jarník, V., O jistém problému minimálním, 1930, Práce Moravské Přírodovědecké Společnosti
[4] Prim, R. C., Shortest connection networks And some generalizations, Noviembre 1957, Bell System Technical Journal
[5] Dijkstra, E. W., A note on two problems in connexion with graphs, Diciembre 1959, Numerische Mathematik

```
Sea corte el conjunto de corte con los ejes que pasan de A a B
Mientras corte ≠ ∅
        Sea e=(a,b) el eje de menor  costo en corte con a ∈ A y b ∈ B
        A = A - {a}
        B = B + {a}
        arbol = arbol + {e}
        Actualizar conjunto de corte
Retornar arbol
```

La lógica del algoritmo es sencilla. Se puede ver que se apoya en la propiedad de corte que demostramos anteriormente. En cada subproblema se elige el menor eje de un determinado conjunto de corte y esto asegura que la elegida pertenezca a un árbol recubridor mínimo. Por otro lado, los nodos alcanzados siempre estarán en el conjunto A. Esto evita que en el corte exista la posibilidad de seleccionar un eje que resulte en un ciclo en la solución que construimos. En conclusión construiremos una solución óptima utilizando este algoritmo.



Tenemos una instancia G como se muestra en la figura. Seleccionemos para iniciar el nodo C. El corte estará conformado por C en un extremo y (A,B,D,E,F) en el otro. Hay tres ejes en el conjunto de corte y el menor de ellos es C-E con costo 2. A continuación tenemos un nuevo corte con (C,E) por un lado y (A,B,D,F) en el otro. Entre los ejes del conjunto de corte seleccionamos E-D con costo 1. Ahora tenemos (C,D,E) y (A,B,F). En el conjunto de corte elegimos A-E con costo 1. Luego elegimos el eje B-E. En la última iteraciones seleccionamos el eje C-F con costo 2. El árbol conseguido es mínimo y tiene un costo de 8.

La complejidad del mismo depende de las estructuras utilizadas para la implementación. Es clave utilizar una buena estructura para poder seleccionar el próximo eje de menor valor y para consultar el grafo. Además pretendemos ejecutar la iteración principal no más

de n veces. La propuesta es utilizar una lista de adyacencias para representar el grafo. Con él podemos acceder de forma eficiente los ejes de un nodo determinado. El espacio requerido para esta estructura **O**(n+m). Podemos ir registrando en un vector si un determinado nodo pertenece al conjunto A o B. El costo de la observación y actualización es **O**(1). El costo de almacenamiento es **O**(n). Los ejes seleccionados los podemos ir almacenando en una lista con costo de inserción **O**(1) y de almacenamiento **O**(n). Finalmente podemos utilizar una cola de prioridad con actualización de clave para registrar los ejes en el corte. Se almacenará únicamente un eje por nodo en el conjunto B. En caso de estar previamente un nodo y encontrarse un nuevo eje de menor costo se decrementará su clave (f_{dec_clave}) con el nuevo valor. Cada nodo se ingresará una única vez ($f_{insertar}$) y se puede actualizar hasta n veces. La cantidad total de actualizaciones de clave estará vinculada a la cantidad total de ejes en el grafo. Se realizan como mucho "n" obtener el ejé mínimo (f_{obt_min}). Además del costo del eje se debe almacenar a cual corresponde. para luego guardarlo en el árbol resultante. El análisis de complejidad es similar al presentado en Dijkstra. En el siguiente grafo se indica según la elección de la cola de prioridad.

Cola prioridad	f_{obt_min}	$f_{insertar}$	f_{dec_clave}	Complejidad
Heap Binario	**O**(logn)	**O**(logn)	**O**(logn)	**O**((n + m)*logn)
H. Binomial	**O**(logn)	**O**(1)	**O**(logn)	**O**((n + m)*logn)
H. Fibonacci	**O**(logn)	**O**(1)	**O**(1)	**O**(n*logn + m)

El siguiente algoritmo que analizaremos es conocido como **algoritmo de Kruskal**. Este fue propuesto en 1956 por Joseph Kruskal[6]. Al igual que el anterior corresponde a un algoritmo greedy. Inicialmente considera a los nodos del grafo a analizar como nodos sin conexiones. Partimos de un conjunto de n árboles de un solo nodo cada uno dentro de un bosque. De forma iterativa selecciona ejes del grafo con el objetivo de ir uniendo árboles. La elección de los ejes se realiza comenzando por los ejes de menor costo a los más costosos. Realiza una evaluación del eje viendo si sus extremos pertenecen a árboles diferentes. Si la respuesta es afirmativa incluye el eje y nos quedamos con un nuevo bosque con un árbol

[6] Joseph Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, 1956, Proceedings of the American Mathematical Society.

menos. En caso contrario se desecha el eje pues su inclusión generaría ciclos en un árbol. El proceso se repite hasta que el bosque está compuesto únicamente por un árbol o que todos los ejes se hayan analizado. En el primer caso el resultado será el árbol recubridor mínimo. En el segundo caso el grafo original no era conexo y nos quedamos con los árboles recubridores de cada una de sus componentes conexos.

Árbol recubridor mínimo - Kruskal

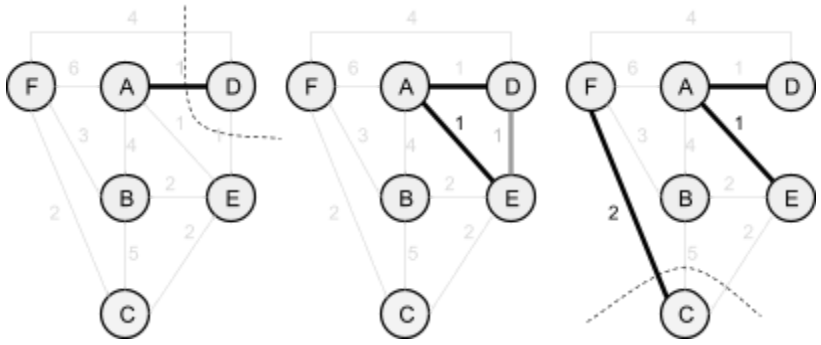
```
Sea Ejes el conjunto de los ejes de G
Sea bosque un conjunto de árboles
Insertar a bosque con cada nodo de G como un arbol
Mientras Ejes  = ∅ y cantidad de arboles en bosque >1
        Sea e=(a,b) el eje de menor costo en Ejes con x,y nodos del grafo
        Ejes = Ejes - {e}
        Sea A arbol que incluye a
        Sea B arbol que incluye b
        Si A≠ B
                Sea C arbol resultante de unir de A y B con eje e
                bosque = bosque - {A} - {B} + {C}
Retornar bosque
```

Con Kruskal tomamos un camino diferente para armar el árbol recubridor mínimo. aunque en el fondo se sigue utilizando la misma propiedad de corte para asegurar la optimalidad. Pero vamos paso a paso. Cuando seleccionamos el primer eje e_1 este será el de menor costo en todo el grafo (o en caso extremo existirán otros de igual costo). Este eje une a dos nodos que son vistos como árboles. Los llamaremos A y B. Sin temor a equivocarnos podemos trazar un corte del grafo donde tenemos por un lado A y el resto de los nodos que no están en A (entre ellos claramente B). El conjunto de corte resultante serán todos los ejes que contengan un nodo en cada extremo. Entre ellos e_1 que al ser el menor eje de todo el grafo también lo es del conjunto. Por la propiedad de corte pertenece al árbol recubridor mínimo. El algoritmo entonces incluye ese eje y conforma un nuevo árbol con A y B.

El algoritmo continúa iterando. Analicemos el paso i. Se obtiene el eje e_i. Todos los anteriores tienen igual o menor costo. Si ambos extremos de e_i pertenecen al mismo árbol, incluirlo genera un ciclo. Para romper ese ciclo y continuar con el menor costo posible tenemos que eliminar un eje que lo conforma. Ciertamente por el criterio de ordenamiento

e_i reúne las condiciones de eliminación. Por lo tanto se verifica la conveniencia de descartar este eje. Por el contrario, si ambos extremos del eje pertenecen a árboles A, B diferentes entonces el algoritmo lo agrega. Tracemos nuevamente un corte donde tenemos A de un lado y los demás árboles del otro. Vemos que los únicos ejes en el conjunto de corte deben ser aquellos que van del árbol A al resto de ellos. Entre ellos e_i. No hay ejes seleccionados que pasen de un conjunto al otro (sino tendríamos nodos de A en ambos conjuntos). Lo que equivale a decir que no hay ejes en el conjunto de corte con coste menor a e_i. En definitiva este, por propiedad de corte debe pertenecer al árbol recubridor mínimo. Demostrando lo acertado del procedimiento de Kruskal

El algoritmo tiene dos criterios de finalización. Cuando existe un solo árbol en el bosque sabemos que los n nodos están comunicados. Como no hay ciclos por condiciones del algoritmo implica que hemos seleccionado n-1 ejes. Sumado que probamos que las elecciones de los ejes son pertenecientes a la solución óptima podemos afirmar que tendremos el árbol recubridor mínimo de la instancia. Si el algoritmo finaliza por quedarnos sin ejes para analizar sabemos que el grafo no era conexo. Pero también que para cada uno de los componentes conexos obtenidos hallamos un árbol recubridor mínimo.



Al comenzar tenemos un bosque con 6 árboles. Además tenemos a los ejes a analizar ordenados de menor a mayor costo: A-D (costo 1), A-E (1), D-E (1), C-F (2), B-E (2), C-E (2), B-F (3), A-B (4), D-F (4), B-C (5) y A-F (6). Comenzamos con el eje A-D. Queda seleccionado puesto que sus extremos pertenecen a árboles diferentes. Se puede armar un corte con D de un lado y el resto del otro. Y se observa que la propiedad de corte se cumple en la elección. Continuamos con A-E que también se selecciona. Al momento de evaluar D-E se descarta por que sus extremos ya pertenecen al mismo árbol y su inclusión generaría un ciclo. Se continúa seleccionando C-F. En

ese punto se puede observar que tenemos 3 árboles en el bosque. Continuamos incluyendo B-E. Se termina de formar el árbol recubridor mínimo al incluir C-E con costo total 8.

Nuevamente de acuerdo a las estructuras utilizadas será la complejidad obtenida. Para construir y mantener los ejes ordenados podemos usar una cola de prioridad. En este caso solo insertamos al inicio y extraemos el mínimo iterativamente. Un heap binario será eficiente. Por otro lado, necesitamos una estructura para verificar que nodos se encuentran en el mismo árbol y cuáles no. En este caso es útil utilizar la estructura de datos para conjuntos disjuntos (disjoint-set data structure también conocida como union–find data structure). En esta estructura - implementada de forma eficiente - podremos unir conjuntos o determinar si ciertos elementos pertenecen o no a un mismo conjunto en $\mathbf{O}(\alpha(n))$. α(n) es la función inversa de ackerman y a efectos prácticos para n de uso razonables podemos considerarlos como una operación de tipo constante. Con ella en vez de crear y mantener árboles mantendremos conjuntos de nodos. Para saber cómo están unidos esos nodos podemos simplemente guardar en una lista los ejes seleccionados. Teniendo en cuenta estas estructuras el algoritmo inicia la cola de prioridad ingresando los m ejes con un costo total de $\mathbf{O}(m \log m)$. Luego recorre todos los ejes obteniendo el de mínimo costo de la cola. Para los extremos del mismo verifica si están en diferente conjunto en el disjoint-set en $\mathbf{O}(\alpha(n))$. Si es el caso se agrega el eje a la solución en $\mathbf{O}(1)$ y se unen los conjuntos de pertenencia de los nodos en el disjoint-set en $\mathbf{O}(\alpha(n))$. En tiempo totales podemos expresarlo como $\mathbf{O}(m \log m + m (\log m + \alpha(n) + 1 + \alpha(n)))$ que en forma simplificada es $\mathbf{O}(m \log m)$. En cuanto a la complejidad espacial del algoritmo, el disjoint requiere un espacio en $\mathbf{O}(n)$, la lista de ejes seleccionados es también $\mathbf{O}(n)$ y la cola de prioridad es $\mathbf{O}(m)$.

Existen otros métodos para construir árboles recubridores mínimos. Entre ellos el de Borůvka[7] o el Reverse-delete_algorithm (también propuesto por Kruskal en la misma publicación).

Merece tomarnos unos minutos más para analizar un problema análogo pero opuesto al aquí planteado. La mayoría de los problemas permite observarlos. En este caso podemos enunciar el problema del árbol recubridor máximo. La diferencia fundamental es que buscamos maximizar el costo total de los ejes seleccionados. Si el enunciado del problema

[7] Borůvka, Otakar, O jistém problému minimálním, 1926, Práce Mor. Přírodověd

además asegura que todos los ejes tienen un costo negativo entonces cualquiera de los algoritmos vistos resuelve de forma eficiente y óptima el problema. Sin embargo, si tenemos únicamente ejes positivos, dado el enunciado de nuestro problema vemos que la solución es seleccionar todos los ejes. Obviamente no conseguiremos un árbol sino el grafo mismo. Para que tenga sentido reformulamos agregando como restricción explícitamente que el resultado tiene que ser un árbol que contenga todos los nodos del grafo. Llegados a este punto se puede ver que (incluso teniendo ejes positivos o negativos) con los algoritmos vistos también podemos resolver este problema. Los tendremos que adaptar modificando el criterio de elección o el orden de recorrido de los ejes. Todo esto queda en evidencia demostrando que la propiedad de corte funciona para el eje más costoso buscando un árbol de mayor costo.