Trabajo Práctico I: Técnicas Algorítmicas

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Morales Pessacq, Joaquín	430/20	joacomp489@gmail.com
Prida, Fabrizio Gabriel	31/20	pridafabrizio@gmail.com
Santesteban, Martín	397/20	martin.p.santesteban@gmail.com
Muñoz, Thomas Nahuel	1821/21	thomm88@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Contenidos

1. Ejercicio 1

1.1. Presentación del problema

Dada una grilla de m filas y n columnas (donde $2 \leq m, n \leq 8$), se desea saber de cuántas formas puede un robot recorrer todas las posiciones dentro de la grilla comenzando y terminando el recorrido en las posiciones $(0, 0)$ y $(0, 1)$ respectivamente (denotando a (x, y) como la posición en la fila x, columna y). Además, el robot debe asegurarse de encontrarse en ciertas posiciones (de ahora en más *checkpoints*) en cada cuarto del recorrido. Es decir, si los checkpoints son p_0, p_1 y p_2, el robot deberá encontrarse en p_i en el i-ésimo cuarto del trayecto, lo que es equivalente a decir que tiene que haber recorrido $\lfloor (n \cdot m) \cdot 0{,}25 \cdot i \rfloor$ celdas.
Por otra parte, vale aclarar que el robot solo puede dar un paso a la vez, para los lados, adelante y atrás.

1.2. Resolución del problema

Podemos comenzar generando todos los caminos posibles usando *backtracking*, partiendo de la posición inicial y chequeando si el camino recorrido es válido una vez que el robot no tiene más posiciones válidas adyacentes. Con esta idea en mente, generamos un árbol de llamados recursivos donde la raíz corresponde a solo haber visitado a la celda inicial y los nodos hijos denotarán el haber dado un paso en alguna de las posiciones válidas. Luego, una vez procesado el árbol, podemos recorrerlo para poder ver todos los caminos posibles. Es importante notar que cada nivel del árbol corresponderá con el haber dado cierta cantidad de pasos en la grilla, y consecuentemente tendremos un árbol de altura $n \cdot m$. Cada nodo interno representará un camino parcial, y cada nodo hoja representará el final de alguno, ya que el robot no podrá avanzar en ninguna posición válida y por lo tanto el nodo no podrá tener hijos.
Notemos que con esta aproximación se computará una cantidad prácticamente infinita de hojas, ya que entre ellas se encontrarán caminos que se quedan sin pasos válidos antes de completar toda la grilla y otros que llegan a checkpoints (o que pasan por la posición final) en el momento incorrecto.
Como consecuencia de esto, se implementaron podas para poder detener el cómputo de caminos que ya sabemos que serán inválidos una vez completada la ejecución. Por lo tanto, nos ahorramos generar las hojas si el camino que hicimos hasta ahora ya es inválido.

1.3. Podas

Poda 1: En la celda indicada en el momento equivocado

El objetivo de esta poda es poder descartar todos los caminos en los que se visita un checkpoint en el momento incorrecto. Por lo tanto, cada vez que se da un paso en alguna posición, se chequea si la posición actual es un checkpoint (o si es la posición final). En caso de serlo, vemos si la cantidad de pasos que dimos coincide con la cantidad de pasos que deberíamos haber hecho para estar en el checkpoint actual. Si no coincide, significa que todos los trayectos que se deriven del camino actual serán inválidos.
Notemos que la complejidad de hacer este chequeo es constante: sólo debemos comparar si la cantidad de pasos es igual a una constante en ciertas posiciones.

Poda 2: En rango

Esta poda busca detener la ejecución si no es posible alcanzar alguno de los checkpoints en la cantidad de pasos necesaria. Dicho de otra forma, si la posición actual está muy lejos de algún checkpoint (o la celda final), detenemos la ejecución. Más rigurosamente, determinamos esta condición si se cumple que para todo checkpoint p la distancia Manhattan entre p y la posición actual es menor a la resta entre la cantidad de pasos necesarios para visitar a p y la cantidad de pasos dados. Esto tiene complejidad O(1), ya que se trata de una resta y una comparación, hecha cuatro veces.
Esta poda nos permite descartar todos los caminos en los que se recorren demasiadas celdas antes de visitar los checkpoints, y así no tener que esperar a llegar a alguno para descartar el camino utilizando la Poda 1.
La figura 1 expone un caso en el que el robot visitó siete celdas, pero se encuentra demasiado lejos del primer checkpoint. Podemos ver que nuestro robot debería encontrarse dentro de las celdas rojizas en el séptimo paso, ya que son las que se encuentran a menos de tres celdas de distancia del checkpoint.

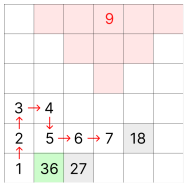

Figura 1. *Camino inválido: el robot se encuentra demasiado lejos del primer checkpoint.*

Poda 3: Generando particiones

El objetivo es detener la ejecución cuando no se puede visitar todo el tablero debido a que el robot haya bloqueado el acceso a alguna parte del mismo. Esto se puede dar al momento de elegir visitar una posición tal que se genere una partición del tablero.

Para llevar a cabo esta poda, la primer idea fue chequear si se particiona el tablero en dos al completar alguna fila o columna, y luego chequear si hay alguna celda sin visitar en ambas lados de la grilla. Esto es debido a que si el robot parte el tablero en dos, tendrá que seguir el recorrido por una de las dos particiones, bloqueando el acceso a la segunda. Luego, si existe una celda sin visitar en ambas particiones, sabemos que por lo menos una celda se ha vuelto inalcanzable y podemos detener el cómputo de este subárbol.

Luego, en un llamado en el que nos encontremos en la posición (i, j) vamos a tener que chequear si la i-ésima fila fue visitada por completo, y en caso de serlo iterar sobre toda la matriz para encontrar las dos celdas sin visitar, y luego repetir el proceso con la j-ésima columna. La primer imagen de la figura 2 muestra un caso donde las celdas azules marcan una partición y las amarillas otra. Notemos que se estudian las particiones debido a que la fila 3 fue visitada completamente.

Aunque esta poda resultó muy útil, vale la pena destacar que sólo detecta particiones cuando se completan filas y columnas, pero podrían generarse otro tipo de particiones sin necesidad de que las mismas se completen. Por otra parte, al completar una fila/columna se debe iterar sobre toda la matriz en búsqueda de dos celdas sin visitar, y esto tiene complejidad $O(n \cdot m)$.

Con el afán de poder detectar cuándo el robot genera una zona de la grilla inalcanzable independientemente de si la fila o columna actual fueron visitadas por completo, surgió la siguiente idea.

El tablero inicialmente no tiene posiciones visitadas y solo el robot deja un camino continuo de celdas visitadas, por lo tanto, si llega a tener enfrente una celda visitada significa que hay un camino entre la que tiene enfrente y la actual que encierra una zona (una "burbuja") formando una partición. Luego, si al encerrar la burbuja tiene dos celdas sin visitar en los lados significa que formó particiones inalcanzables. Por lo tanto, podemos detener la ejecución si la celda actual tiene en dos lados opuestos celdas sin visitar, y en los otros dos celdas visitadas, como expone la segunda figura de 2.

Este método es mejor que el anterior debido a que nos deja encontrar burbujas al formarlas, independientemente de la fila y columna actual. Además, esto tiene complejidad constante (solo debemos iterar sobre las cuatro celdas adyacentes). Más allá de que esta poda sea muy eficiente, no nos deja detectar *todas* las burbujas que se formen. Sólo detectará aquellas que se forman cuando el robot se choca de frente, pero no cuando se forma encerrándola por los lados, como muestra el último ejemplo de la figura 2.

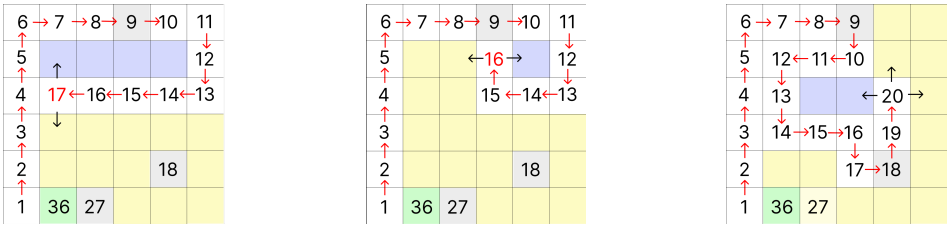

Figura 2. *Las primeras dos figuras muestran casos en los que se generan particiones, la ultima muestra un caso en el que nuestra poda no las detecta.*

Poda 4: Celdas restantes

Esta es la poda más eficiente y a la vez la más compleja. El objetivo es ver la cantidad de celdas que quedan por visitar partiendo de la posición final, deteniendo la ejecución si nos encontramos con celdas ya visitadas por el robot. Por lo tanto, si el robot particionó la grilla en dos partes, el algoritmo contará la cantidad de celdas sin visitar que se encuentran en la partición a la que pertenece la posición final. Luego, si este es distinto a la resta entre las celdas totales y las recorridas por el robot, significa que no existe un camino válido y se debe podar la rama.

Esto lo llevamos a cabo con backtracking, comenzando la ejecución en la posición final. El algoritmo es relativamente simple, si la celda en la que estamos no fue visitada ni por el robot ni por la poda, sumamos uno al resultado, y hacemos recursión sobre las celdas adyacentes. Si fue visitada no la contamos y no hacemos recursión sobre las adyacentes, asegurándonos de contar una vez todas las celdas que quedan por visitar partiendo de la posición final. Si el número de celdas que contó la poda es igual al total de celdas menos las recorridas, significa que si el robot no se encerró entonces existe todavía una forma de recorrer lo que resta de la grilla visitando todas las celdas. Exponemos un ejemplo de este caso en la primer figura de 3. Si el número de celdas contadas es menor, significa que se generó una partición sí o sí, ya que el algoritmo no pudo contar celdas restantes partiendo de la posición final. La figura 3 expone ambos casos. Las celdas rojizas son las que cuenta la poda, y las azules las que se pierde.

Figura 3. *A la izquierda podemos ver un ejemplo del primer caso, a la derecha exponemos un ejemplo del segundo.*

Con respecto a la complejidad, este algoritmo de backtracking genera un árbol de llamados recursivos, donde los nodos internos solo deben sumar 1 en caso de no haber visitado la celda actual, mientras que las hojas detienen la recursión. Por lo tanto, procesar nodos tiene complejidad constante. Luego, el calculo de complejidad será $O(\#nodos \cdot O(1))$. Para calcular la cantidad de nodos, pensemos que cada subinstancia hace llamados recursivos a los nodos adyacentes únicamente. Notemos que cada celda tiene cuatro celdas adyacentes. Además, cada subinstancia realizará estos llamados una única vez, a paritr de ahi se la considera visitada. Luego, las celdas adyacentes a la misma pueden llegar a llamarla de nuevo, pero al ya haber sido visitada tendrá complejidad O(1) ejecutarlo. Luego, como hay $n \cdot m$ celdas, y cada subinstancia corresponde a una celda que el algoritmo visitará como máximo 4 veces, la complejidad de la poda es $O(n \cdot m)$.

1.4. Complejidad

Para el cálculo de complejidad del algoritmo, debemos tener en cuenta que cada nivel del árbol corresponde con cierta cantidad de pasos(todo nodo en el i-ésimo nivel corresponde a un camino que dió i pasos). Notar que todo nodo tiene como máximo 4 nodos hijos, donde uno si o si corresponde a una celda visitada. Luego, para el calculo de complejidad, por ahora nos abstraemos de este cuarto hijo hoja ya que se diluye con la notación O grande. Por lo tanto, sabemos que tenemos $\sum_{i=1}^{m \cdot n} 3^i$ nodos, donde $3^{m \cdot n}$ son nodos hoja. La complejidad de las hojas en este nivel es O(1), ya que lo único que deben hacer el robot es ver si se encuentra en la posición final, habiendo dado la cantidad correcta de pasos. Sin embargo, la complejidad de los nodos internos es esquivalente a la sumatoria de las podas. Como vimos antes, todas tienen complejidad constante menos la poda 4, que tiene complejidad $O(n \cdot m)$. Luego, la complejidad final del algoritmo es $O((\sum_{i=1}^{m \cdot n-1} 3^i) \cdot m \cdot n + 3^{m \cdot n}) = O(3^{m \cdot n} \cdot m \cdot n)$.

2. Ejercicio 2

2.1. Descripción del problema

Se tienen n aspersores instalados en un terreno de l metros de largo por w metros de ancho. Para cada aspersor se provee la distancia desde el extremo izquierdo del terreno, lo cual nos permite conocer su posición puesto que los

mismos se ubican en el centro horizontal del terreno. Además, se provee el radio de operación de cada uno.
El objetivo es determinar el mínimo número de aspersores que debemos encender para cubrir con agua todo el terreno.

2.2. Resolución del problema

2.2.1. Primeras ideas

El punto de partida para resolver el problema surgió con el hecho de tener que usar un enfoque goloso. Lo primero que nos preguntamos no tenía que ver con *qué* decisión golosa tomar, sino con *cómo* se debería reducir el problema una vez tomada la decisión. La alternativa más natural fue considerar ir cubriendo el terreno de izquierda a derecha. Comenzamos queriendo cubrir el intervalo $[0, l]$ del terreno, y en cada paso quisiéramos elegir un aspersor que cubra (por lo menos) desde el extremo inferior del intervalo hasta un cierto punto $0 \leq x \leq l$. Al elegir dicho aspersor, tendremos cubierto el terreno hasta x, por lo que en los pasos siguientes deberemos cubrir el intervalo $[x, l]$ del terreno, eligiendo un subconjunto de los aspersores restantes.

Sin embargo, al pensarlo de esta manera estamos asumiendo que sabemos de antemano el intervalo del terreno que cada aspersor cubre por completo, cuando en realidad la información que se nos proporciona es la posición y radio de los mismos. Si para cada aspersor tuviéramos un rango $[d_i, h_i]$ tal que se cubren todos los puntos (x, y) del terreno con $d_i \leq x \leq h_i \ \wedge \ -w/2 \leq y \leq w/2$, entonces la estrategia mencionada anteriormente podría ser una opción viable.

Para definir el intervalo del terreno que es completamente cubierto por un aspersor, vamos a calcular los puntos donde se intersectan el círculo (definido por la posición y radio) y las rectas horizontales que definen los límites del terreno. La Figura 4 permite ver gráficamente cómo se puede pasar de una descripción de los aspersores en círculos a una en base al intervalo máximo cubierto $[d, h]$.

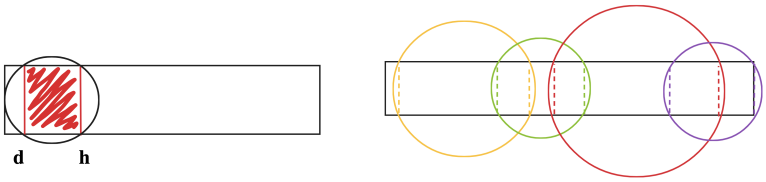

Figura 4. *A la izquierda se muestra el rango máximo de cubrimiento de un aspersor en base a su intersección con los límites del terreno. A la derecha la transformación de círculos a rectángulos de cubrimiento.*

A partir de esta idea es que vamos a pre-procesar la información de entrada para pasar de tener como información de los aspersores un par (l_i, r_i) (*i.e,* distancia del extremo izquierdo del terreno al centro del aspersor y radio respectivamente) a tener un par (d_i, h_i) que denota el intervalo que el i-ésimo aspersor cubre completamente. En caso de que el radio de algún aspersor sea tal que no se llegue a cubrir ningún intervalo por completo, procedemos a descartarlo puesto que no nos permite incrementar el intervalo de terreno cubierto y aún así se vería incrementada la cantidad de aspersores utilizados.

2.2.2. Decisión golosa

Sea $A = \{(d_1, h_1), ..., (d_n, h_n)\}$ el conjunto de aspersores, donde cada aspersor i es representado por un par (d_i, h_i) que indica el intervalo del terreno cubierto completamente.

La idea del algoritmo goloso es que vaya cubriendo el terreno de izquierda a derecha. Sea entonces $WG(A, k, l)$ una función recursiva que toma un conjunto de aspersores A, una distancia parcial k y la longitud del terreno l. El objetivo es que retorne la mínima cantidad de aspersores necesarios para cubrir por completo el intervalo $[k, l]$ del terreno.

En base a la semántica de la función descrita es que queremos tomar una decisión golosa para ir achicando el problema, hasta que eventualmente lleguemos a que $k = l$, de forma tal que habremos cubierto todo el terreno. Para reducir el intervalo, debemos elegir un aspersor para extender la región cubierta. Si hemos cubierto el terreno hasta la posición k, entonces el conjunto de aspersores candidatos a ser elegidos es $candidatos(k) = \{(d, h) \in A : \ d \leq k \leq h\}$. Cualquier otro aspersor que no pertenezca a este conjunto no es útil para extender el terreno cubierto, pues estaría dejando un hueco sin cubrir.

Ahora bien, aquí entra en juego la decisión golosa: entre los candidatos, nos queremos quedar con aquel que llegue a cubrir lo más a la derecha posible, es decir, entre los que forman parte de $candidatos(k)$, aquel que tenga la segunda componente más grande. La idea detrás de esta decisión es que, si en cada paso maximizamos el terreno cubierto entonces estaremos minimizando la cantidad empleada de aspersores .

Luego, definimos la función recursiva de la siguiente manera:

$$WG(A,k,l) = \begin{cases} 0 & si \quad k \geq l \\ \bot & si \quad k < l \ \wedge \ cubreMas(A,k) = (\bot,\bot) \\ 1 + WG(A - \{cubreMas(A,k)\}, cubreMas(A,k).second, l) & cc \end{cases}$$

Aquí usamos la función $cubreMas(A,k)$, que toma un conjunto de aspersores y una longitud k, y devuelve aquel aspersor (d_i, h_i) tal que $d_i \leq k \leq h_i$ que además cumple que no existe otro aspersor (d_j, h_j) tal que $d_j \leq k \leq h_i < h_j$. En caso de que no exista ningún aspersor bajo esas condiciones retorna $(\bot, \bot)$. Es decir, retorna el aspersor que más hacia la derecha cubre a partir de k, si es que existe.

2.2.3. Algoritmo para resolver el problema

A fin de desarrollar un algoritmo eficiente que resuelva el problema planteado, lo primero que hacemos es ordenar crecientemente los aspersores de acuerdo a, en primer lugar, el límite inferior de los intervalos que cubre cada uno, y en caso de que este valor coincida se desempata por el límite superior de sus intervalos.

Una vez realizado el ordenamiento mencionado, utilizamos el algoritmo `wateringGrass(A, k, l, i)` que toma la secuencia ordenadada de aspersores A, la longitud parcial ya cubierta k, la longitud del terreno l y un índice i que indica el último aspersor utilizado de la secuencia. En cada paso el algoritmo se encarga de encontrar, a partir del aspersor de la posición $i+1$, aquel que cubra desde una longitud menor o igual a k lo más a la derecha posible. Si se encuentra un aspersor en la posición $j > i$ que cubre hasta una longitud $k' > k$, basta retornar $1 + wateringGrass(A, k', l, j)$. Ahora bien, si no existiese un aspersor en dichas condiciones es porque no se puede cubrir el terreno, y en ese caso se retorna -1 para indicar esta situación. De la misma manera, si el llamado recursiva se indefine, también se retorna -1.

En base a este algoritmo, podemos utilizar `wateringGrass(A, 0, l, -1)`, de forma tal que se comience a recorrer la secuencia A desde el índice $i + 1 = 0$.

2.2.4. Análisis de complejidad del algoritmo

En primer lugar tenemos el costo de ordenar crecientemente n aspersores. Utilizando la función `std::sort` con una comparación que cuesta $O(1)$ tenemos que el costo para ordenar la secuencia es de $O(n \cdot log(n))$.

Pasando a la función `wateringGrass`, en peor caso son necesarios todos los aspersores para cubrir por completo el terreno. En esta situación tendremos $O(n)$ nodos, uno por cada llamado. En cada llamado recorremos la secuencia de aspersores, pero como la recorremos una única vez, tenemos que el costo de hacer todos los llamados y recorrer la secuencia es $O(n)$.

De esta manera, la complejidad total queda $O(n \cdot log(n)) + O(n) = O(n \cdot log(n))$.

Observación. En nuestra implementación anterior pagábamos $O(n)$ en *cada* llamado recursivo puesto que no ordenábamos la secuencia de aspersores y teníamos que hacer una búsqueda lineal en cada caso. Ahora nos ahorramos eso, utilizando el ordenamiento de los aspersores.

2.2.5. Correctitud del algoritmo

Sea $G_k = e_1, ..., e_k$ la *secuencia de elecciones golosas*. En primer lugar definimos el conjunto de aspersores no utilizados hasta el momento como $A_k = A - \bigcup_{i=1}^{k} \{e_i\}$ ($A_k = A - G_k$ como abuso de notación), donde A es el conjunto inicial de aspersores. En base a esto, se define $G_0 = \emptyset$, y se define $G_k = G_{k-1} \bullet e_{k+1}$ (donde $\bullet$ denota la concatenación) tal que $e_{k+1} = (d_{k+1}, h_{k+1})$ pertenece a A_k y cumple que $d_{k+1} \leq h_k \leq h_{k+1}$ y además, dado un aspersor $(d', h') \in A_k$ tal que $d' \leq h_k \leq h'$ se cumple que $h_{k+1} \geq h'$.

Es decir, partimos del problema inicial $WG(A, 0, l)$ con $G_0 = \emptyset$. Tras las primeras k elecciones golosas tenemos cubierto el terreno hasta la longitud h_k y nos quedan sin utilizar los aspersores del conjunto A_k. Definimos G_{k+1} a

partir de G_k, a la cual le agregamos el aspersor del conjunto A_k tal que cubra desde una longitud menor o igual a h_k y se extienda lo más lejos posible hacia la derecha, sin dejar terreno descubierto.

Ahora bien, si tenemos la secuencia de elecciones golosas $G_k = e_1, ...e_k$ con $h_k < l$ y no existe ningún aspersor e_i que cumpla $d_i \leq h_k \leq h_i$, definimos extender la secuencia como $G_{k+1} = G_k \bullet (-1, -1)$, indicando así que la misma no está definida. Es decir, dada una secuencia G_i que no está bien definida, sabemos que cubre todo el terreno hasta la longitud $h_{i-1} < l$ y que no existe ningún aspersor para seguir extendiendo de forma continua el intervalo de cubrimiento.

Como en cada paso se va agregando un aspersor a la solución, quisiéramos ver que en caso de que el algoritmo no se indefina, la secuencia resultante tiene cardinal mínimo, es decir, es una solución óptima. Para la demostración que sigue vamos a considerar G_r como la secuencia golosa que se obtiene al cubrir todo el terreno, con r la cantidad de aspersores empleados.

Teorema. *Si G_r está bien definida entonces es una solución óptima.*

Demostración. Para ver esto vamos a demostrar que toda secuencia golosa G_k bien definida se puede extender a una solución óptima. Lo demostramos por inducción en k.

$$P(k) \equiv \textit{Si } k \leq r \textit{ y } G_k \textit{ está bien definida, entonces es extensible a una solución óptima.}$$

Caso base: $k = 0$**.** Trivial pues $G_0 = \emptyset$ y a partir de la misma se puede extender a cualquier solución, en particular a una óptima.

Paso inductivo: $P(k) \Rightarrow P(k+1)$**.** Asumimos $k+1 \leq r$ pues en otro caso es trivial. Sea $G_k + 1 = e_1, ..., e_k, e_{k+1}$ una secuencia golosa. Por hipótesis inductiva sabemos que $G_k = e_1, ..., e_k$ se puede extender a una solución óptima. Sea $X = x_{k+1}, ..., x_r$ dicha extensión, ordenada de acuerdo a su primer componente (*i.e,* el límite inferior del intervalo que cubren). Notemos ahora que X es una solución óptima para el problema $WG(h_k)$ (*i.e,* cubre el terreno que le falta cubrir a G_k, con la mínima cantidad de aspersores). Notemos también que $X \subseteq A_k$ (*i.e,* utiliza únicamente aspersores que todavía no fueron utilizados). Tomemos $x_{k+1} = (d', h')$ el primer aspersor de X. Sabemos que $d' \leq h_k \leq h'$ pues al extender G_k no se deja terreno descubierto. Consideremos ahora el aspersor e_{k+1} para el cual sabemos que también se cumple $d_{k+1} \leq h_k \leq h_{k+1}$. Se presentan dos posibilidades:

- $x_{k+1} = e_{k+1}$. Si coinciden, entonces directamente podemos tomar $X' = X - x_{k+1}$ como extensión de G_{k+1}.
- $x_{k+1} \neq e_{k+1}$. En este caso, como e_{k+1} es el que cubre más terreno hacia la derecha desde la longitud h_k, sabemos que $h_{k+1} \geq h'$ (*i.e,* el aspersor dado por la elección golosa cubre hasta una longitud mayor o igual que lo que cubre el aspersor x_{k+1}). En consecuencia, si tomamos $X' = X - \{x_{k+1}\} \cup \{e_{k+1}\}$ obtenemos otra extensión para G_k que resulta en una solución óptima. Y se puede afirmar que dicha solución es óptima puesto que al tomar ahora como primer aspersor de X' a uno que cubre más terreno hacia la derecha que x_{k+1}, sabemos que el terreno va a ser cubierto completamente, y además la cantidad de aspersores utilizados no aumentará.

  Como ahora tenemos la extensión $X' = e_{k+1}, x_{k+2}, ..., x_r$, podemos tomar la secuencia golosa G_{k+1} y considerar como extensión a $X'' = X' - \{e_{k+1}\}$.

En cualquier caso, se ve que es posible extender la secuencia golosa a una solución óptima, por lo que el paso inductivo queda entonces demostrado.

Tras haber probado $P(k)$, sabemos que la elección golosa G_r se puede extender a una solución óptima S. Como G_r cubre todo el terreno con r aspersores, sigue que $r \geq \#(S)$. Pero por $P(k)$ sabemos que $r \leq \#(S)$ puesto que podemos extender G_r. En consecuencia, $\#(G_r) = \#(S)$, por lo que podemos afirmar que si G_r está bien definida entonces es una solución óptima. □

Lo que ahora resta ver es el caso en el que la secuencia golosa no esté bien definida.

Teorema. *Si la secuencia golosa no está bien definida entonces el terreno no puede ser cubierto por un subconjunto de los aspersores.*

Demostración. Sea $G = e_1, ..., e_k, (-1, -1)$ una secuencia golosa que no está bien definida. Por definición, sabemos que con la misma se cubrió el terreno hasta la longitud $h_k < l$. Pero además, si consideramos el conjunto de los aspersores restantes A_k, se cumple que no existe ningún aspersor (d', h') tal que $d' \leq h_k \leq h'$. En consecuencia, el terreno que va desde la longitud h_k y hasta la primer longitud abarcada por los aspersores de Ak no será cubierto. Por lo tanto, el terreno, en su totalidad, no podrá ser cubierto por los aspersores que se tienen. □

2.2.6. Equivalencia entre el problema planteado y el problema resuelto

En lo que sigue vamos a querer ver que al haber pasado de pensar los aspersores como círculos a pensarlos en base al intervalo del terreno que cubren por completo, el problema no cambia.

Dado un terreno de l metros de largo por w de ancho, sea $C = \{(l_1, r_1), ..., (l_n, r_n)\}$ el conjunto de n aspersores pensados como círculos, donde (l_i, r_i) indica la distancia desde el extremo izquierdo del intervalo al centro del aspersor y el radio del mismo respectivamente. A partir de C definimos $R = \{(d_1, h_1), ...(d_n, h_n)\}$ como el conjunto de los mismos aspersores pero expresados a partir del intervalo que cubren completamente.

Teorema. *Sean $C = \{c_1, ...c_k\} \subseteq C'$, $R = \{r_1, ...r_k\} \subseteq R'$ los conjuntos de aspersores equivalentes, expresados como círculos y rectángulos respectivamente. C cubre todo el terreno $\iff$ R cubre todo el terreno.*

Demostración. $\implies$) Sea $p = (x, y)$ un punto del terreno. Si p se encuentra en alguno de los intervalos $[d_i, h_i]$ que se deducen a partir de los círculos $c_i \in C$, claramente ocurrirá que p será cubierto por el rectángulo $r_i \in R$.

Consideremos entonces los puntos que no son parte del intervalo cubierto completamente por un círculo. Dado el círculo $c_i = (l_i, r_i) \in C$ que cubre completamente el rectángulo $r_i = (d_i, h_i) \in R$, tomamos los puntos $p = (x, y)$ del terreno tal que $h_i < x \leq d_i + r_i \ \wedge \ y > \sqrt{r_i^2 - (x - d_i)^2}$. Todos esos puntos pertenecen al intervalo $[h_i, d_i + r_i]$ y no son cubiertos por el círculo c_i. Ahora bien, como C cubre todo el terreno, sabemos que debe existir otro círculo $c_j = (l_j, r_j) \in C$ tal que cubra todos los puntos definidos anteriormente. En particular, c_j debe cubrir a un punto de la forma $p' = (h_i + \lambda, w/2)$ con $\lambda > 0$ (*i.e* un punto en el extremo superior del terreno). En consecuencia, tomando λ suficientemente pequeño, sigue que $d_j \leq h_i$ para que de esta manera el círculo c_j (que cubre el intervalo $[d_j, h_j]$) cubra al punto p'. En la Figura 5 podemos observar esta situación. En particular se puede ver que el hecho de que el círculo c_j cubra al punto p' implica que $d_j \leq h_i$.

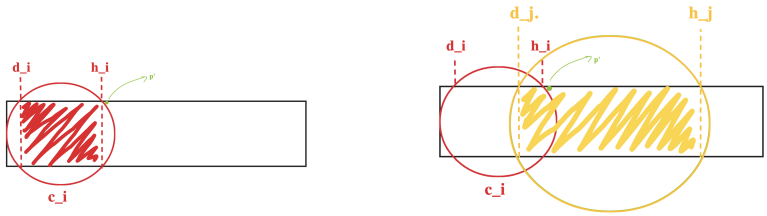

Figura 5. *A la izquierda observamos un círculo c_i sin cubrir a un punto $p = (h_i + \lambda, w/2)$. A la derecha se incluye el círculo c_j que sí lo cubre.*

Luego, hemos demostrado que si un punto del terreno es cubierto por algún círculo $c_i \in C$, también lo es cubierto por un rectángulo $r_i \in R$.

$\impliedby$) Al ser R un recorte en rectángulos del terreno cubierto por C, sabemos que todo el terreno cubierto por R está incluido en el terreno cubierto por C. Luego, si R cubre todo el terreno, necesariamente C también. □

3. Ejercicio 3

3.1. Descripción del problema

Este ejercicio es igual al anterior, salvo una variación que se le agrega: ahora cada aspersor, además de su radio y distancia al extremo izquierdo del terreno, tiene asociado un costo. El objetivo es encontrar el costo mínimo para cubrir todo el terreno.

3.2. Resolución del problema

Para resolver este problema vamos a utilizar ciertas ideas desarrolladas para el Ejercicio 2. En particular, vamos a considerar que tenemos como información de entrada, para cada aspersor, el intervalo del terreno que cubren completamente. Así, vamos a pensar cada aspersor como una estructura con tres valores: *.desde* y *.hasta* que indican el intervalo máximo que cubre por completo el aspersor, y .costo indicando el costo del mismo.

En base a esta información para cada aspersor, definimos A como el conjunto que contiene los n aspersores. Para lo que sigue vamos a considerarlo ordenado crecientemente de acuerdo al atributo *.desde*. En caso de que dos aspersores

coincidan en este valor, se desempata por el valor *.hasta* de cada uno, también en orden creciente. En particular, esto nos será útil a la hora de definir la función recursiva. Además, vamos a descartar aquellos aspersores cuyo radio sea menor a $w/2$, puesto que los mismos no cubren ningún intervalo del terreno y en consecuencia no serán utilizados.

3.2.1. Función recursiva

Semántica.
En lo que sigue vamos a definir $costoMin : \mathbb{N} \to \mathbb{Z}$ como la función recursiva para resolver el problema. Su semántica es la siguiente:

> $costoMin(i)$ indica el costo mínimo para cubrir por completo el intervalo $[A_i.desde, l]$ del terreno, utilizando únicamente los aspersores $A_i, A_{i+1}, ..., A_n$.

Definiendo la función recursiva.
La idea es que el intervalo que se cubre por completo se vaya extendiendo de derecha a izquierda (recordando que por el orden de los aspersores vale que $A_k.desde < A_i.desde$ si $k < i$). Luego, el caso base lo definiremos cuando se considere el último aspersor ($i = n - 1$), de forma tal que si dicho aspersor llega a cubrir hasta el final del terreno, entonces el valor que debe retornar la función es su costo. Si por el contrario no llegase a cubrir hasta el final se retorna ∞, pues podemos asegurar que no se puede cubrir por completo el terreno.

Para el caso recursivo, tomamos un aspersor $i < n-1$. Asumimos que tenemos resueltos todos los subproblemas para $j \in [i+1, n-1]$, es decir, sabemos el costo mínimo para cubrir el terreno a partir de la posición $A_j.desde$ y utilizando únicamente los aspersores a partir el j-ésimo, para $i < j < n$. Para definir $costoMin(i)$, debemos considerar todos los casos posibles de cubrir el intervalo de interés:

- **El intervalo se cubre uniendo el i-ésimo aspersor a otro conjunto de aspersores.** Dado un subproblema ya resuelto $costoMin(j)$ (con $j > i$) tal que se cubre el terreno desde la posición $A_j.desde$, en caso de que se cumpla que $A_i.hasta \geq A_j.desde$, entonces podremos considerar $A_i.costo + costoMin(j)$ como candidato a ser $costoMin(i)$, puesto que se cubre todo el intervalo $[A_i.desde, l]$.
- **El i-ésimo aspersor cubre por completo el intervalo de interés.** Podría ocurrir que el i-ésimo aspersor de por sí solo cubra hasta el final del terreno. En ese caso, entonces $A_i.costo$ es candidato a ser $costoMin(i)$.
- **El intervalo es cubierto por un subconjunto de aspersores ya considerados.** Si $A_i.desde = A_{i+1}.desde$, tenemos que $costoMin(i+1)$ es tal que ya cubre el intervalo $[A_i.desde, l]$, pero mirando únicamente los aspersores $A_{i+1}..., A_{n-1}$. Si este fuese el caso, entonces también será candidato a ser $costoMin(i)$.

Luego, entre todos estos costos candidatos, la respuesta será el mínimo de todos ellos.

En base a las ideas descritas se define la función recursiva de la siguiente manera:

$$costoMin(i) = \begin{cases} A_i.costo & si \quad i = n-1 \ \wedge \ A_i.hasta \geq l \\ \infty & si \quad i = n-1 \ \wedge \ A_i.hasta < l \\ min^\star\{x : x \in \ agregoI(i) \ \bigcup \ siCubroTodo(i) \ \bigcup \ siYaSeCubre(i)\} & cc \end{cases}$$

donde:

$$agregoI(i) = \{A_i.costo + costoMin(k) \ / \ i < k < n \ \wedge \ A_i.hasta \geq A_k.desde\}$$

$$siCubroTodo(i) = \begin{cases} \{A_i.costo\} & si \quad A_i.hasta \geq l \\ \emptyset & cc \end{cases}$$

$$siYaSeCubre(i) = \begin{cases} \{costoMin(i+1)\} & si \quad A_i.desde = A_{i+1}.desde \\ \emptyset & cc \end{cases}$$

Aquí, la función $min^\star$ retorna ∞ en caso de que el conjunto que recibe fuese vacío.

Conjuntos definidos para escribir la función recursiva
Para el caso recursivo se definen tres conjuntos que hacen referencia a las formas posibles de cubrir el intervalo de interés, descritas anteriormente:

1. «agregoI(i)» está formado por valores que se obtienen de agregar el costo del i-ésimo aspersor a $costoMin(j)$ (con $j > i$) de forma tal que se cubra por completo el intervalo de interés.
2. «siCubroTodo(i)» contempla el caso de que el i-ésimo aspersor cubra por completo el intervalo $[A_i.desde, l]$. En caso afirmativo, entonces $A_i.costo$ es candidato.
3. «siYaSeCubre(i)» da cuenta del caso en que el intervalo ya es cubierto por un subconjunto de aspersores mayores al i-ésimo, es decir, $costoMin(i+1)$ es un costo que ya cubre desde la longitud $A_i.desde$.

Luego, la unión de estos conjuntos representa los costos de todas las posibles formas de cubrir por completo el intervalo $[A_i.desde, l]$. La función retorna, entre todos ellos, el valor mínimo. Si el conjunto fuese vacío, entonces no hay manera de cubrir el intervalo, por lo que en ese caso se retorna ∞.

3.2.2. Correctitud de la función recursiva

La idea general para ver que la función recursiva definida resuelve el problema viene de notar que, al resolver $costoMin(i)$, de alguna manera se están considerando todas las opciones posibles para cubrir el intervalo $[A_i.desde, l]$. Para definir $costoMin(i)$ miramos, en primer lugar, todos los intervalos $[j, l]$ tal que $A_i.desde < j \leq A_i.hasta$. Dichos intervalos son tal que al agregar el i-ésimo aspersor, se cubre *por completo* el intervalo $[A_i.desde, l]$. Además, si queremos minimizar el costo total, vamos a querer que los costos para cubrir los intervalos $[j, l]$ sean mínimos. Esos costos para dichos intervalos con los pertenecientes al conjunto «agregoI(i)». Es decir, la función recursiva contempla todos los casos en los que cubrimos el intervalo $[A_i.desde, l]$ a partir de tener cubierto un intervalo menor $[j, l]$ y agregar el i-ésimo aspersor.
Ahora bien, también podría pasar que el costo mínimo no se obtenga en base a agregar el i-ésimo aspersor al costo mínimo de un intervalo menor. En ese caso, o bien el i-ésimo aspersor cubre hasta el final del terreno, o bien el de interés ya es cubierto con costo mínimo. Estas dos opciones son consideradas por los conjuntos «siCubroTodo(i)» y «siYaSeCubre(i)» respectivamente.
Es decir, la función recursiva contempla todas las posibilidades para definir el costo mínimo: los casos en los que el i-ésimo aspersor es usado (ya sea porque cubre todo el terreno o porque se puede "*unir*" a un intervalo menor sin dejar huecos), y, por otra parte, el caso en que el i-ésimo aspersor no es utilizado (puesto que con los aspersores siguientes ya se cubre todo el terreno de interés). De todas las posibilidades, la función se queda con el mínimo.

3.2.3. Superposición de problemas

Para que valga la pena memoizar, quisiéramos ver que se cumple la propiedad de superposición de problemas.
Sea n la cantidad de aspersores. Entonces, $\#instancias = O(n)$, puesto que el único parámetro de la función recursiva i es tal que $0 \leq i < n$. Ahora, para calcular la cantidad de llamados recursivos, consideremos el caso en que $agregoI(i)$ tiene un elemento, lo cual va a funcionar como cota inferior. En este caso, para resolver $costoMin(i)$ se van a hacer, desde dicha instancia, 2 llamados (uno para $agregoI(i)$ y otro para $siYaSeCubre(i)$). Luego, como para cada i tenemos dos llamados, para resolver el problema tendremos en total 2^n llamados recursivos.
Como $\lim_{n\to\infty} \frac{\#llamadosRecursivos}{\#instancias} = \lim_{n\to\infty} \frac{\Omega(2^n)}{O(n^2)} = \infty$, podemos afirmar que vale la pena memoizar para resolver el problema, ya que se están llevando a cabo varios llamados recursivos para los mismos subproblemas.
Observando que hay superposición de problemas suponiendo que $agregoI(i)$ realiza un único llamado recursivo, podemos asegurar que en todos los demás casos en los que dicho conjunto tiene más de un elemento (*i.e,* realiza más de un llamado) también valdrá la pena memoizar, puesto que se harán más llamados recursivos que los calculados previamente.

3.2.4. Memoización

Consideramos como estructura de memoización un vector M de tamaño n, con n la cantidad de aspersores. El mismo es inicializado con un valor que indica que está indefinido al comienzo. Luego, a medida que se resuelven las subinstancias $costoMin(i)$, se va almacenando el resultado en $M[i]$. De esta manera, tras haber calculado por primera vez un subproblema, podemos acceder a su resultado en tiempo constante, sin tener que volver a calcularlo explícitamente.

3.2.5. Algoritmo para resolver el problema

Como ya se mencionó, lo primero que se hace es convertir la descripción de los aspersores a una que se base en el intervalo que cubre por completo cada uno.
Luego, se ordena el conjunto de aspersores como se mencionó anteriormente. Con esto obtenemos una secuencia A de aspersores ordenada, que será utilizada por el algoritmo principal.
Definimos el algoritmo $costoMinimo(M, A, i, l)$ que implementa la función recursiva definida anteriormente. Sus parámetros son: la estructura de memoización M, el vector de aspersores ordenado A, el índice que representa la subinstancia a resolver i, el largo del terreno l.
El caso base sigue los lineamientos de la función recursiva. El caso recursivo también, pero chequeando el estado de M. Si no hay almacenado un valor válido, se procede a calcularlo por primera (y única vez) de la siguiente manera: se genera un vector de valores *candidatos*, que son aquellos pertenecientes a los conjuntos «agregoI(i)», «siCubroTodo(i)», «siYaSeCubre(i)».. Luego, se retorna el mínimo entre todos estos valores (o -1 si el conjunto es vacío), actualizando la estructura de memoización.

3.2.6. Análisis de complejidad del algoritmo

Repasemos lo que hace el algoritmo, indicando la complejidad de las operaciones. Antes que nada, se realiza lo siguiente:

- *se pre-procesa la información de los aspersores.* Esto se hace en costo $O(n)$, teniendo en cuenta que el costo para describir a un aspersor en base al intervalo que cubre completamente es $O(1)$.
- *se ordenan los aspersores en forma creciente.* Al utilizar la función `std::sort`, el costo de esto es $O(n \cdot log(n))$.

Ahora pasemos a las operaciones propias de `costoMinimo` para resolver $valorMin(i)$:

- *se realizan a lo sumo $n - 1 - i$ llamados recursivos*, lo cual tiene un costo $O(n)$, teniendo en cuenta que realizar un llamado es $O(1)$.
- *se busca el mínimo entre todos los candidatos a ser la respuesta.* Como tenemos $O(n)$ candidatos, al recorrer los mismos en búsqueda del mínimo pagamos $O(n)$.

En consecuencia, el costo para resolver un subproblema es $O(n)$. Como tenemos $O(n)$ subinstancias, sigue que el costo del algoritmo `costoMinimo` es $O(n) \cdot O(n) = O(n^2)$. Si además tenemos en cuenta el pre-procesamiento y el ordenamiento de los aspersores, el costo final resulta ser $O(n) + O(nlog(n)) + O(n^2) = O(n^2)$.