DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP1 de AED3

4 de mayo de 2024 — Algoritmos y Estructuras de datos III

Estudiante	LU	Correo electrónico
Lucas Mas Roca	122/20	lmasroca@gmail.com
Luciana Skakovsky	131/21	lucianaskako@gmail.com
Mateo Cantagallo	143/21	mateocantagallo@gmail.com
Alan Yacar	174/21	alanroyyacar@gmail.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
http://www.exactas.uba.ar

1. Introducción

En este informe se encontrarán las explicaciones y demostraciones de correctitud y complejidad de los algoritmos propuestos para la resolución de los problemas planteados. Se describirá primero una breve explicación del problema (en muchos casos de forma más abstracta) y luego se desarrollara la solución pensada.

2. Problema 1

En este problema se nos presenta una grilla de M filas $\times N$ columnas con $2 \leq M, N \leq 8$. En esta grilla debemos encontrar la cantidad de caminos que existen pudiendo solo moverse de casilla en casilla en una de las 4 direcciones cardinales que cumplan las siguientes condiciones:

1. La casilla inicial es la (0,0).
2. La casilla final es la (0,1).
3. Pasa por todas las casillas exactamente una vez.
4. Se cuenta con 3 coordenadas pasadas como parámetro que corresponden a la ubicación en la cual se debe estar habiendo recorrido un cuarto, la mitad y tres cuartos de las casillas respectivamente.

El algoritmo deberá devolver el número de caminos que cumplen las condiciones tomando como entrada los valores de M, N y las coordenadas intermedias.

3. Solución 1 : Backtracking

Comenzamos analizando el problema y notando que se trata de un problema de conteo combinatorio. Nuestro espacio de soluciones factibles consta de todos los caminos posibles dentro de la grilla, una vez generado este espacio de soluciones, solo faltaría contar cuantas de estas representan una solución válida chequeando las condiciones antes mencionadas. Puede verse como esta idea de fuerza bruta resultaría muy costosa. Si generamos los caminos considerando las 4 posibles direcciones en cada paso puede verse que el tamaño del espacio de estas soluciones será 4^{mxn}.

Por esta razón debemos utilizar la técnica de backtracking para contar la cantidad de caminos válidos posibles existentes que cumplen con todas las condiciones para devolver una respuesta, pudiendo cortar con el análisis de un camino en el momento en el que deja de ser válido. Esto último lo vamos a obtener tras aplicar las siguientes podas, que cortaran la generación de un camino en cuanto este ya no pueda resultar en un camino completo válido.

Empezamos ubicándonos sobre la posición (0,0) de la grilla, y vamos a ir generando todos los recorridos posibles, para esto debemos en cada paso llamar recursivamente a la función sobre las 4 posiciones posibles sobre las cuales podemos movernos. Para considerar un camino como válido, debemos recorrer toda la grilla sin repetir posiciones ni salirnos de ella, además de pasar por todos los checkpoints (incluyendo la casilla final) en el momento adecuado.

Estas condiciones pueden ser revisadas durante la construcción de los recorridos y de esta forma descartar recorridos que no son válidos de forma más eficiente, en esto consisten las podas. Para esto, vamos a utilizar una matriz de booleanos de igual dimensión que el tablero, que nos va a permitir

saber si una posición ya fue visitada o no, y de esta forma podar los recorridos que repiten posiciones de forma eficiente.

Adicionalmente, descartaremos todos los recorridos ni bien nos salgamos del tablero y al pasar por un checkpoint si no realizamos la cantidad de movimientos requeridos para ese checkpoint. Si estamos en la cantidad de movimientos requeridos para algún checkpoint y no estamos en ese checkpoint, podemos descartar ese recorrido también. Además, agregamos que si nos encontramos en una posición y nuestra distancia al próximo checkpoint es mayor a la cantidad de pasos que tenemos para llegar al mismo, descartamos ese camino. Luego de estas podas iniciales llegamos a la conclusión de que se requerían algunas más inteligentes para cumplir con la cota temporal. Por esto implementamos las siguientes podas.

La primera de ellas constaba de chequear al llegar a cada borde las casillas correspondientes a los bordes adyacentes de manera antihoraria. Por ejemplo, al llegar a alguna casilla que se encuentre en el borde derecho significa que el camino llego de un borde al otro, por lo tanto, si quedan casillas en la parte de arriba del tablero estas no se pueden visitar (porque no hay camino para ir y volver). Esto quiere decir que al llegar a algún borde derecho todos los bordes derechos de arriba deben haber sido recorridos.

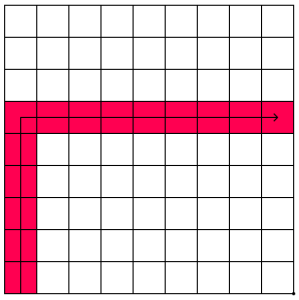

Figura 1: Se ve acá que las casillas encima de la del borde nunca podrán ser alcanzadas.

Análogamente, realizamos los chequeos correspondientes en cada borde.

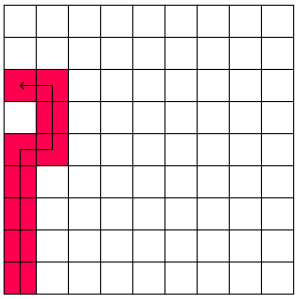

Figura 2: Al llegar al borde izquierdo chequeo que todos los bordes izquierdos de abajo hayan sido visitados para no tener esta situación.

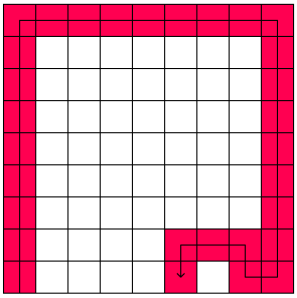

Figura 3: Al llegar al borde de abajo chequeo que todos los bordes de abajo a la derecha hayan sido visitados para no tener esta situación.

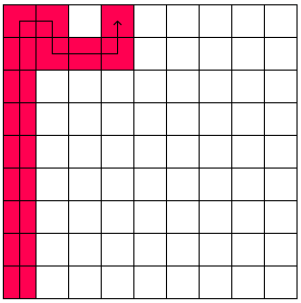

Figura 4: Al llegar al borde de arriba chequeo que todos los bordes de arriba a la izquierda hayan sido visitados para no tener esta situación.

Finalmente, realizamos una poda en caso del recorrido bloqueé el tablero, como no podemos repetir posiciones y no podemos salirnos del tablero, si el recorrido llega a una posición en la cual no podemos movernos o nos quedan posiciones que nunca van a poder ser cubiertas, debemos descartar ese recorrido. Para esto implementamos una función que en cada paso chequea si se cruza con el camino ya recorrido y quedaron bloques sin visitar en el medio, en este caso corta la recursión.

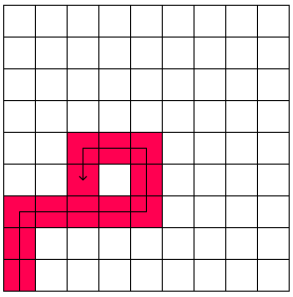

Figura 5: Este tipo de casos son los que se intentan evitar.

4. Problema 2

En este problema se cuenta con un campo de longitud l y ancho w y n aspersores. Cada uno de estos aspersores se encuentra en el centro horizontal del rectángulo y se nos provee con el radio que cubren y su posición con respecto al borde izquierdo. Se busca conocer cual es la cantidad mínima de aspersores que se deben prender para poder cubrir todo el campo, si es posible.

5. Solución 2: Algoritmo Greedy

5.1. Idea:

Primero, debemos tener en cuenta que el radio del aspersor no es lo que realmente nos importa, como todos los aspersores están ubicados verticalmente sobre la mitad del ancho del $(\frac{w}{2})$, nos interesa el área que cubre del terreno de forma completa (en otras palabras, nos importa el rectángulo que cubre hasta la parte superior del terreno). Debemos tener en cuenta que los aspersores que tienen radio menor o igual a $\frac{w}{2}$ debemos sacarlos, ya que no podremos cubrir ninguna sección del terreno usando estos aspersores. Veamos un ejemplo: Podemos ver como usando Pitágoras, podemos conseguir

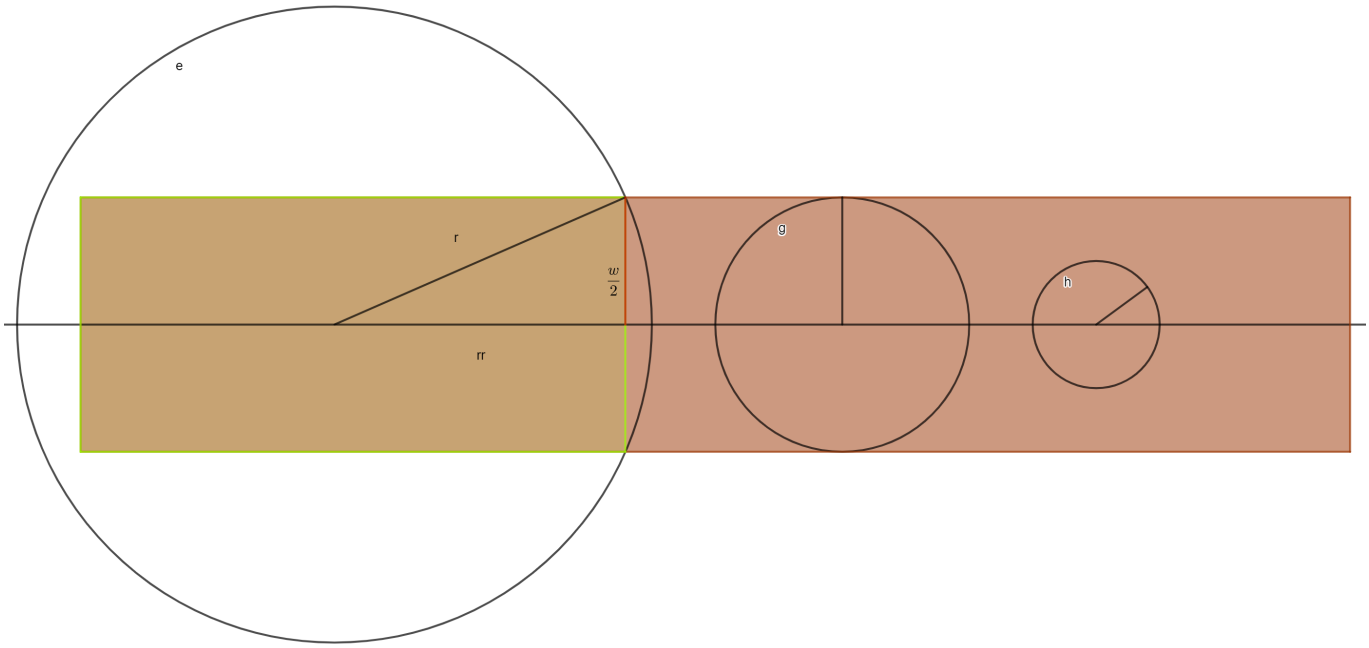

Figura 6: Representación de aspersores

la longitud de rr. Notemos que no es necesario trabajar con el rectángulo, sino que podemos usar solamente el segmento que representa la longitud del rectángulo sobre el terreno, ya que su ancho será igual.

$$c_1^2 + c_2^2 = h^2$$

$$rr^2 + (\frac{w}{2})^2 = r^2$$

$$rr^2 = r^2 - (\frac{w}{2})^2$$

$$rr = \sqrt{r^2 - (\frac{w}{2})^2}$$

Luego, si el centro del aspersor es x, el segmento que nos importa va desde $x - rr$ hasta $x + rr$. Podemos descartar también todos los segmentos que se encuentran completamente fuera del terreno.

Una vez tenemos todos los aspersores representados como segmentos (habiendo filtrado aquellos aspersores que no aportaban a la resolución del problema) pasamos a resolver el problema con el algoritmo. La idea de la solución es empezar por el lado izquierdo (longitud 0 en la recta que va desde 0 hasta l) e ir prendiendo el aspersor que más área cubre hacia la derecha (primero nos fijamos entre todos los aspersores que tenemos cuál es el que más área cubre hacia la derecha y luego lo prendemos) desde el punto izquierdo actual, luego actualizamos el segmento que nos falta cubrir (el cual empieza siendo desde 0 hasta l) para que su extremo izquierdo sea el extremo derecho del aspersor que acabamos de prender, repetimos este proceso hasta cubrir todo el segmento (en caso de que haya solución, si no se puede cubrir todo, para algún segmento nos encontraremos con que no podemos cubrirlo desde la parte izquierda con ninguno de los aspersores que tenemos).

Podemos ver como la idea de la solución propuesta es una estrategia golosa, ya que en cada paso (cada aspersor que prende) toma el aspersor que más área cubre sin considerar las futuras combinaciones posibles de aspersores ni el área a cubrir luego del paso. Notemos que otra idea posible para una estrategia golosa podría ser prender siempre el aspersor que más área nueva cubre (con área nueva nos referimos a área que no esté ya cubierta por un aspersor que se prendió en una iteración anterior) hasta cubrir todo el terreno (en caso de que sea posible, si no se puede veremos que en alguna iteración ninguno de los aspersores restantes aportan área nueva), pero esta idea no siempre funciona! Esta estrategia golosa no es un algoritmo goloso, sino que es una heurística.

Para hacer esto de forma más eficiente, luego de traducir los aspersores a segmentos y filtrarlos, los ordenamos (primero por su extremo izquierdo, luego por el extremo derecho en caso de empate en el extremo izquierdo). De esta forma, al iterar sobre los aspersores podemos empezar a buscar desde donde termino la búsqueda anterior hasta el primer aspersor que tiene su intervalo a la derecha del extremo izquierdo del segmento libre. Al hacer esto, conseguimos complejidad lineal ($O(n)$) en el algoritmo (la complejidad de la solución va a estar dada por la complejidad del sort) en vez de complejidad cuadrática, ya que solo iteramos una vez sobre el conjunto de aspersores.

5.2. Demostracion:

Para demostrar la correctitud de este algoritmo debemos probar dos cosas. Primero que si nuestro algoritmo no devuelve solución (da -1) entonces no existe una solución válida al problema bajo los parámetros de entrada dados. Segundo es demostrar que la solución devuelta es la óptima, es decir, no existe una solución que cubra toda el área pero que haya prendido menos aspersores.

1. Nos quedamos sin aspersores para prender y no llegamos a cubrir del lado derecho.

2. Existe al menos un segmento entre aspersores el cual no podemos cubrir con ninguno de los aspersores que tenemos disponibles.

1 - En este caso, el algoritmo goloso prendió todos los aspersores que cubrían la máxima área a derecha en cada iteración, pero no tiene más aspersores para prender y no llego a cubrir toda el área. Con lo cual existe un segmento entre la derecha del último aspersor prendido y l el cual no puede ser cubierto por ningún aspersor, por lo tanto, en ese caso no hay solución.
2 - Para este caso, en alguna iteración del algoritmo nos vamos a encontrar con que entre todos los aspersores disponibles, la máxima área a derecha posible en esa iteración es 0, con lo cual existe al menos un segmento entre 0 y l que no puede ser cubierto por ningún aspersor, con lo cual en ese

caso no existe solución.

Queda probar que la solución devuelta por el algoritmo goloso es óptima, en caso de que exista solución. Esto se resume en demostrar que dada cualquier solución al problema, tendrá al menos la misma cantidad de aspersores prendidos. Para esto definimos las siguientes secuencias:

$$G_k = g_0, ..., g_k; Y_s = y_0, ..., y_s$$

Siendo G_k la secuencia de intervalos dados por el algoritmo goloso e Y_k una secuencia de intervalos que resuelve el problema, ordenados de izquierda a derecha, y como son solución, el i-ésimo elemento cubre a izquierda el área de los intervalos anteriores. Sabemos que los intervalos están ordenados de izquierda a derecha por su extremo izquierdo, por lo tanto, el área restante por cubrir luego de considerar hasta el i-ésimo intervalo va a ser $l - \sum_{j=0}^{i} y_j$[1]. El problema estará resuelto cuando este área sea menor o igual a cero, también sabemos que nuestro algoritmo no agregara más intervalos luego de que este área restante sea menor o igual a cero.

Si comparamos nuestras dos secuencias, para determinar que $k \leq s$ (cosa que garantizaría que G_k es óptima), basta con demostrar que $l - \sum_{j=0}^{i} y_j$ no sea menor o igual a cero para algún $i \leq k$. Por inducción probemos que $\sum_{j=0}^{i} g_j \geq^2 \sum_{j=0}^{i} y_j$

$$HI : \sum_{j=0}^{i} g_j \geq \sum_{j=0}^{i} y_j$$

Caso base: $i = 0$ nuestro algoritmo elige de todos los intervalos que cubren a izquierda el intervalo que cubre más a la derecha (por lo tanto, va a cubrir más área que cualquier otro intervalo que podamos elegir).

Paso inductivo: Queremos considerar el $(i + 1)$ intervalo de la secuencia.

$$\sum_{j=0}^{i+1} g_j \geq \sum_{j=0}^{i+1} y_j$$

$$\sum_{j=0}^{i} g_j + g_{i+1} \geq \sum_{j=0}^{i} y_j + y_{i+1}$$

Por HI:

$$g_{i+1} \geq y_{i+1}$$

Pero sabemos por HI también que el y_{i+1} cubre a izquierda el segmento de $\sum_{j=0}^{i} g_j$ porque cubre el de $\sum_{j=0}^{i} y_j$ que es menor o igual. Como cubre ese segmento, entra entre los candidatos del algoritmo goloso en esa iteración y como es candidato $g_{i+1} \geq y_{i+1}$ que era lo que queríamos ver.

Como $\sum_{j=0}^{i} g_j \geq \sum_{j=0}^{i} y_j$ para todo i el área restante de la secuencia Y en la i-ésima iteración es mayor o igual a la de G, por lo tanto, no puede ser cero para $i < k$.

[1]Definimos $\sum_{j=0}^{i} y_j$ como el área cubierta por todos los intervalos desde y_0 hasta y_i, es decir, sera el intervalo desde el extremo izquierdo de y_0 hasta el extremo derecho del que mas cubra entre ellos.

[2]Definimos el $\geq$ entre áreas como la comparación entre los extremos derechos de las áreas.

6. Problema 3

Muy similarmente al problema anterior se cuenta con un campo y aspersores, pero adicionalmente al caso anterior también se cuenta con el precio de cada aspersor. En este caso lo que se quiere es conseguir el precio mínimo de cubrir todo el campo en caso de que sea posible.

7. Solución 3: Programación Dinámica

Para empezar debemos realizar el mismo proceso que en el problema anterior, es decir, filtrar los aspersores que no aportan a la solución del problema (los que tienen radio menor o igual a $\frac{w}{2}$) y traducirlos a segmentos de la misma forma. Luego, ordenamos los segmentos por su extremo izquierdo (esto nos servirá más adelante). Sea a_i el i-ésimo aspersor, su costo asociado c_i y sus extremos izquierdo y derecho i_i y d_i respectivamente.

La idea será la siguiente: Cada aspersor a_i puede estar prendido, en cuyo caso se debe sumar su costo c_i, o apagado. Puede verse, similar al problema de la mochila (resuelto en clase) que el problema de calcular el mínimo costo de con una instancia de n aspersores y l área puede reducirse a tomar el mínimo entre las subinstancias:

- Tomar los $n-1$ aspersores y mantener la misma área l a cubrir. Este sería el caso en el que no prendo el aspersor a_1.
- Tomar los $n-1$ aspersores, sumar el precio del aspersor n y tomar $l - intervalo(n)$. Este sería el caso en el que si prendo el aspersor a_1.

Formalizando todo esto nos queda la siguiente función recursiva:

$$f(i,j) = \begin{cases} 0 & \text{si } i = n+1 \wedge d_j \geq l \\ \infty & \text{si } i = n+1 \wedge d_j < l \vee d_j < i_i \\ min(f(i+1,j), c_i + f(i+1, argmax(d_i, d_j))) & \text{caso contrario} \end{cases}$$

Siendo $f(i,j)$ el costo mínimo de cubrir el segmento $[d_j, l]$ usando los aspersores $a_i, a_{i+1}, \ldots, a_n$. Siendo $f(0,0)$ el llamado que resuelve el problema, teniendo $c_0 = d_0 = 0$ y $i_0 = -\infty$, adicionalmente si al resolver el problema el resultado es ∞ debemos devolver -1 ya que no existe forma de cubrir todo el campo.
Los casos base de la recursión son:

- Si ya recorrí todos los aspersores y logre cubrir toda el área, entonces el costo es 0.
- Si ya recorrí todos los aspersores y no logre cubrir toda el área, entonces no es no existe solución que cubra todo el campo. Particularmente realizamos un chequeo que en caso de estar en el medio de la recursión y no poder cubrir alguna área intermedia devolver infinito.

Notamos que la función tiene solamente $n \times n$ instancias posibles. Porque i va desde 0 hasta $n+1$ y j va desde 0 hasta $max(0, i-1)$. Más allá de esto, la función recursiva requiere de 2^n llamados recursivos, lo que indica que hay superposición de subproblemas. Debido a esto, para evitar recalcular una misma subinstancia utilizamos una estructura de memoización.

Como dijimos antes, contamos con n^2 posibles llamados entonces utilizamos una matriz m de $\mathbb{Z}^{(n)\times(n)}$ siendo $m[i][j] = f(i,j)$. Teniendo esta estructura podemos ver que la complejidad del algoritmo será $O(n^2)$ porque cada subinstancia se calcula una sola vez y al momento de necesitar el valor, si ya fue calculado, se devuelve el valor en $O(1)$ sin realizar el llamado recursivo.