VNIVERSIDAD NACIONAL
AVTÓNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Minimización de disparos para romper barreras

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
FERNANDO CARBAJAL MORALES

DIRECTOR DE TESIS
DR. JORGE URRUTIA GALICIA
INSTITUTO DE MATEMÁTICAS UNAM

MÉXICO, D. F. NOVIEMBRE 2013

Agradecimientos

Quiero agradecer a mis papás Sara y Luis, a mi hermano Fabro, a mi abuelita Chabelita, y a mi tío Kike por apoyarme incondicionalmente. Me ayudaron todo el tiempo, incluso cuando sólo me escucharon hablar sobre mis problemas.

Agradezco a mi pareja Karla, quien me apoyó siempre. Gracias por tu paciencia, por entenderme, y por siempre estar conmigo.

También agradezco a mis amigos y familiares por motivarme a no detenerme. Gracias Sebas por ayudar a mantenerme en perspectiva.

Del mismo modo, quiero agradecer a los colegios Ángel Matute Vidal y Miraflores. Mis profesores y compañeros son el pilar de mi formación. Gracias Madre Salud Conde Nieto por su valiosa ayuda.

Por supuesto que agradezco también a mi otro hogar, a universidad UNAM. No sólo obtuve conocimiento de mis maestros y compañeros, sino que también obtuve grandes amistades.

También quiero agradecer a mi "chief" y tutor, Jorge. No sólo le agradezco su tiempo y dedicación en este trabajo, sino también su compromiso docente. Sus clases no sólo me sorprendieron y motivaron, también me cambiaron mi forma de pensar totalmente.

Asimismo, agradezco a mis compañeros y al grupo de trabajo de mi tutor. Gracias a mis compañeros por apoyarme, y a mis sinodales por su dedicación y justas correcciones.

También agradezco a todo aquél que influyó en el desarrollo de este trabajo y que no haya mencionado.

Finalmente, gracias a ti lector por tu tiempo, espero te sea útil.

¡Gracias!

Índice general

1. Introducción **2**
1.1. Trabajo relacionado 8

2. Teoría de Gráficas **12**
2.1. Definiciones básicas 12

3. Disparos de potencia infinita **19**

4. Disparos de potencia constante **29**
4.1. Notas de implementación 39

5. Disparos de potencia variable **41**
5.1. NP-completez 41
5.2. Disparos de potencia variable 45

6. Conclusiones **50**
6.1. Trabajo futuro 50

Índice de figuras

1.1. Una configuración que cubre al objeto de interés. 3
1.2. Segmento de recta real. 4
1.3. Segmento de recta horizontal en el plano cartesiano. 4
1.4. Segmentos ajenos en el plano cartesiano. 4
1.5. Segmentos traslapados en el plano cartesiano. 4
1.6. El orden de los disparos es de suma importancia. 5
1.7. Capas de la Tierra. 7
1.8. Juego de Angry Birds. 7
1.9. Karateca rompiendo hielo. 8
1.10. Resumen de resultados para el problema de cubrir segmentos ortogonales. 10
1.11. Definición del subproblema para el problema SSR. 11

2.1. Gráfica 13
2.2. Vértices independientes y vértices adyacentes. 13
2.3. Conjunto independiente (vértices azules). 13
2.4. Conjuntos independientes (vértices azules) 14
2.5. Conjunto completamente conectado dentro de una gráfica. 14
2.6. Gráfica de intersección 15
2.7. Gráfica de intervalos 15
2.8. Gráfica arco-circular 16
2.9. Gráfica arco-circular cortada donde hay intervalos. 16
2.10. Gráfica arco-circular cortada donde no hay intervalos. 17
2.11. Encontrando un conjunto independiente máximo. 17
2.12. Encontrando un conjunto independiente máximo (continúa). . . . 18
2.13. (Continuación) Encontrando un conjunto independiente máximo. 18

3.1. Una barrera y sus extremos. 19
3.2. Una configuración original 20
3.3. Contando las intersecciones en el primer punto 20
3.4. Contando las intersecciones en todos los puntos 20
3.5. Después del primer disparo, si hay más barreras se vuelve a contar y se sigue iterando 20
3.6. Un disparo para cada barrera. 21
3.7. Configuración de barreras inicial. 22
3.8. El algoritmo se posiciona en al punto más a la izquierda. 22
3.9. El algoritmo busca la primera cabeza que queda en la configuración. 22
3.10. El algoritmo destruye todas las barreras que se intersectan con la cabeza encontrada. 22

3.11. Invariante del algoritmo: todas las barreras a la izquierda de un disparo están cubiertas. 23
3.12. Una configuración con diferentes conjuntos independientes máximos. 24
3.13. Conjunto independiente máximo de tamaño tres. 25
3.14. Barrera encerrada. 25
3.15. Configuración de barreras. 26
3.16. Configuración de barreras y el primer vértice de su gráfica de intervalos. 26
3.17. Configuración de barreras y la primera arista de su gráfica de intervalos. 27
3.18. Configuración de barreras y su correspondiente gráfica de intervalos. 27

4.1. Mismas gráficas de intervalos y conjuntos independientes máximos, pero diferente número de disparos para destruirlas. 30
4.2. Configuración en forma de árbol. 32
4.3. Configuración donde reacomodar constantemente tiene complejidad de tiempo cuadrática. 33
4.4. Ejemplo de ejecución donde no se destruye óptimamente la configuración. 33
4.5. Ejemplo de reconfiguración donde se destruye óptimamente. . . . 33
4.6. Ejemplo de reconfiguración en un solo barrido. 34
4.7. Al reconfigurar nos quedan descubiertas las cabezas de las barreras. 36
4.8. Al reconfigurar nos queda la forma de una escalera. 36
4.9. Al disparar sólo se destruye una barrera. 37
4.10. Al disparar sólo se destruye una barrera pero hay otras antes. . . 37
4.11. Al disparar con potencia tres sólo se destruyen dos barreras. . . . 38
4.12. Lo más conveniente es destruir la intersección entre ambos conjuntos al mismo tiempo. 38
4.13. Gráfica no dirigida con un bucle 39
4.14. Matriz de adyacencia correspondiente 39
4.15. Gráfica no digirida y sin bucles 40
4.16. Lista de adyacencia correspondiente 40

5.1. Relación entre P, NP, y NPC si P $\neq$ NP 42
5.2. Algoritmo de reducción de tiempo polinomial. 43
5.3. Función de reducción de tiempo polinomial. 43
5.4. Una configuración donde el conteo ávido funciona. 46
5.5. Una configuración donde el conteo ávido no funciona. 46
5.6. Reducción de un problema a otro. 47
5.7. Reducción del problema de la partición al de disparos de potencia variable. 49

Lista de algoritmos

1. Obtención de un conjunto independiente máximo. 17
2. Minimización de disparos con potencia infinita. 21
3. Construcción de una gráfica de intervalos a partir de una configuración de barreras. 26
4. Reconfiguración de una instancia y destrucción de sus barreras . 32
5. Reconfiguración y minimización de disparos con potencia constante. 35

Capítulo 1

Introducción

Supongamos que existe un objeto que nos interesa mucho obtener, como un tesoro, por ejemplo. Este objeto se encuentra enterrado debajo de muchos obstáculos los cuales nos hacen imposible alcanzarlo mientras estén presentes. Estos obstáculos digamos que son una familia de lápidas horizontales que nos impiden el paso. Para cumplir nuestro objetivo de alcanzar el tesoro debemos destruir estas lápidas. Al conjunto de lápidas lo llamaremos una *configuración*. Para destruir una configuración haremos uso de una pistola de rayos láser. Los disparos provenientes de la pistola los podemos ver como rayos verticales con dirección hacia abajo. Si un disparo toca una lápida la destruye. Cada disparo tiene una potencia (o fuerza) con la que destruye cierta cantidad de lápidas. Es decir, la *potencia* es el número de lápidas (barreras) que un disparo puede destruir. Por ejemplo, un disparo de potencia uno destruirá sólo una lápida y no más. Un disparo de potencia dos destruirá dos lápidas si el disparo atraviesa sucesivamente las dos lápidas. Un disparo de potencia infinita destruirá toda lápida que atraviese. Para aclarar un poco más, de manera práctica podemos pensar en esta potencia como un número igual o más grande al número total de barreras. Por ejemplo, podemos tener una configuración con sólo cinco barreras, y los disparos de potencia infinita los podemos considerar como si tuvieran potencia diez. Entonces ni siquiera las cinco barreras juntas podrían detener uno solo de estos disparos.

En la figura 1.1 mostramos un ejemplo en el cual para destruir los intervalos que modelan las lápidas necesitamos cinco disparos de potencia uno, o tres de potencia dos.

Veremos diferentes variantes del problema. Primero estudiaremos el caso en el que todos nuestros disparos tendrán potencia infinita. En esta variante desarrollaremos un algoritmo el cual con complejidad $O(n \log n)$ nos dice la cantidad mínima de disparos que necesitamos para destruir toda la configuración de barreras exponiendo el tesoro rápidamente. La *complejidad* en tiempo de un algoritmo es el número de pasos que sabemos que le tomará en terminar. La notación de la "gran O" (Big O notation) denota la cantidad máxima de pasos que le puede tomar al algoritmo, es decir, nos da una cota y nos dice qué tan tardado es en el peor caso. Si un algoritmo tiene una complejidad en tiempo

Figura 1.1: Una configuración que cubre al objeto de interés.

de $O(n)$ decimos que tiene una complejidad lineal, y el número de pasos dependerá de qué tan grande sea n. Esta variable, n, es el número de elementos que introducimos. Por ejemplo, si tenemos un programa que ordena un conjunto de elementos, n sería el número de elementos. En nuestro problema, n es el número de barreras de una configuración. Existen diferentes complejidades las cuales las podemos expresar con esta notación. Como ejemplo, pensemos en un algoritmo que simplemente manda imprimir a pantalla un número específico. Su complejidad es de $O(1)$, es decir, en tiempo constante porque en un solo paso termina. Hay otras complejidades, como $O(n^2)$, $O(n^3)$, y $O(n \log n)$, entre otras. En este trabajo nos referiremos siempre a la complejidad en tiempo, excepto cuando indiquemos la complejidad en espacio. Esta última se refiere a la memoria que necesita un algoritmo. En [1] se incluye información más detallada del tema.

Volviendo a nuestro problema, también veremos un caso que es un poco más realista, en donde los disparos tienen una misma potencia constante (ya no infinita). Aquí veremos primero otro algoritmo cuya complejidad es de $O(n \log n)$ en tiempo, y en el peor caso nos da como resultado el doble de disparos que haríamos con el resultado óptimo. Después vemos que si pudiéramos permutar[1] las barreras de la configuración de manera vertical, podemos entonces realmente destruir óptimamente dicha configuración. Para esto haremos uso de otro algoritmo que primero ordena las barreras y después hace disparos óptimos. Finalmente llegaremos al caso más realista en el cual tenemos una colección finita de disparos con potencias diferentes cada uno. Aquí lo que queremos es saber si con el conjunto de disparos que nos es permitido podemos destruir una configuración de barreras que también nos es proporcionada. Es decir, es un problema de decisión.

En este trabajo estudiamos las variantes del problema con el objetivo de dar diferentes estrategias para minimizar el número de disparos utilizados para desenterrar el tesoro. Estudiaremos la versión de este problema en el caso del plano; es decir, nuestro tesoro lo podemos modelar como un segmento de recta, y las lápidas como segmentos horizontales ajenos. Un *intervalo* cerrado real es un conjunto de la forma $\{x \in \mathbb{R} | p \leq x \leq q\}$, donde a p y a q se les llama extremo izquierdo y derecho del intervalo, respectivamente. No está de más hacer notar que un intervalo está sobre la recta real (figura 1.2).

A lo largo de este trabajo utilizaremos los términos *intervalo* y *segmento* de manera indistinta.

[1] No decimos que "movemos" porque se podría confundir con que las barreras pueden chocar entre ellas al moverlas verticalmente.

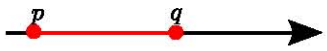

Figura 1.2: Segmento de recta real.

En nuestro problema los intervalos están en el plano, por lo que son conjuntos de la forma $\{(x, y) \in \mathbb{R}^2 | p \leq x \leq q, y = c\}$, donde $c \in \mathbb{R}$ (figura 1.3).

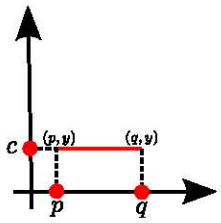

Figura 1.3: Segmento de recta horizontal en el plano cartesiano.

Se dice que los segmentos son *ajenos* cuando la intersección entre ellos en $\mathbb{R}^2$ (el plano) es vacía (figura 1.4).

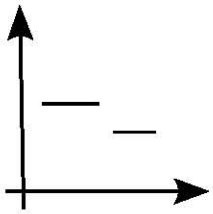

Figura 1.4: Segmentos ajenos en el plano cartesiano.

Se dice que los segmentos se *traslapan* si y sólo si se intersectan sus respectivas proyecciones verticales sobre el eje horizontal (figura 1.5). Abusaremos del lenguaje y diremos que los intervalos *intersectan*, o que son *adyacentes* si se traslapan. Este abuso viene del hecho de que a pesar de que en $\mathbb{R}^2$ no se intersecten, en $\mathbb{R}$ sí lo hacen.

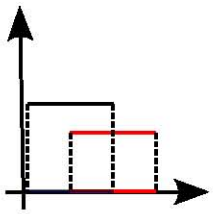

Figura 1.5: Segmentos traslapados en el plano cartesiano.

Veremos también que el orden de los disparos es muy importante. Por ejemplo, en la figura 1.6 supongamos que tenemos disparos de potencia tres, y que decidimos primero destruir las barreras más largas porque nos parece, a primera vista, que así destruiremos las barreras que estorban más. Siguiendo este

criterio, disparamos primero en la tercera columna dos veces, destruyendo la columna junto con las dos barreras grandes. Después disparamos una vez en cada una de las columnas restantes. Sin embargo, si hubiéramos disparado en las columnas en orden de izquierda a derecha nuestros disparos habrían sido menos.

Figura 1.6: El orden de los disparos es de suma importancia.

El contenido de este trabajo está estructurado de la siguiente forma.

En el capítulo 2 introduciremos definiciones, conceptos, y algoritmos que nos ayudarán en el tema. Entendiendo ese capítulo podremos pasar a hacer las demostraciones de los capítulos siguientes.

En el capítulo 3 tenemos el caso de disparos de potencia infinita. En ese capítulo utilizaremos un algoritmo similar al de [2] para destruir todas las barreras de complejidad $O(n \log n)$. El algoritmo de [2] lo veremos en el capítulo 2, y en realidad sirve para encontrar un conjunto independiente máximo. Un *conjunto independiente máximo* es un conjunto tal que los elementos que pertenecen a dicho conjunto no son adyacentes entre sí. Es decir, para nuestro problema el conjunto contendrá como elementos a barreras que no son adyacentes entre sí. De principio esto nos dice que un conjunto independiente máximo nos indica cuántos disparos necesitamos para destruir una configuración. El análisis de esta observación está en el capítulo 3. Para más detalles relacionados con un conjunto independiente máximo ver el capítulo 2.

En el capítulo 4 nos encontramos con los disparos de potencia constante. A diferencia del caso de potencia infinita, aquí tendremos disparos de una potencia k, por lo que un disparo podrá destruir a lo más k barreras suponiendo que todas intersectan entre sí (una misma recta vertical). Primero veremos un algoritmo *ávido* (greedy) para minimizar el número de disparos en esta variante. Un algoritmo de este tipo toma decisiones que localmente son las mejores con la esperanza de encontrar el óptimo global. Hay que notar que los algoritmos de este tipo no siempre dan la mejor solución, sino que dan una solución aproximada a la mejor. El algoritmo ávido que desarrollamos en particular da una solución que en el peor caso hace el doble de disparos del óptimo. Se divide en dos fases. La primera consiste en hacer disparos que no desperdicien potencia, mientras que la segunda toma la configuración "desgastada" por la primera y ejecuta el algoritmo de potencia infinita del capítulo 3. En ese mismo capítulo veremos otros dos algoritmos que podemos usar con la condición de que podemos permutar las barreras verticalmente (es decir, únicamente podemos cambiar su posición vertical). El primero es bueno si tenemos una configuración de barreras muy específica. El último, de complejidad de $O(n \log n)$, acomoda cualquier configuración de forma que disparar con un orden específico es siempre óptimo.

El capítulo 5 muestra el caso donde tenemos una colección finita de disparos, cada uno con una potencia dada, pero esta potencia puede no ser la misma

para todos. Por ejemplo, podemos tener tres disparos: uno de potencia uno, otro de potencia cinco, y el otro de potencia tres. Lo que haremos aquí será una reducción de un problema ya estudiado. Una *reducción* es un algoritmo que transforma un problema en otro ya conocido. Normalmente se aplica para mostrar que el problema que estamos estudiando es por lo menos tan difícil como el conocido (la reducción la hacemos desde el conocido hacia el nuestro). El problema que utilizaremos para hacer la reducción es el "problema de la partición" (Partition problem). Es sabido que es NP-completo.

La abreviación *NP* significa tiempo polinomial no determinístico; NP es el conjunto de todos los problemas de decisión para los cuales si la respuesta "sí" o "no" es verificable en tiempo polinomial ("rápidamente") por una *máquina de Turing determinística*. A grandes rasgos, esta máquina es una máquina teórica la cual es imaginada como una computadora simple, la cual escribe y lee símbolos (uno a la vez) sobre una cinta infinita siguiendo un conjunto de reglas definidas. Por ejemplo, una regla puede ser "si te encuentras en el estado dos y lees una A, cámbiala por una B y muévete a la derecha". Mientras que una máquina de Turing no determinística puede tener diferentes reglas para un mismo estado. Por ejemplo, puede tener la regla que acabamos de mencionar y además la regla "si te encuentras en el estado dos y lees una A, cámbiala por una C y muévete a la izquierda". Se dice que es no determinística porque no se sabe qué regla elegirá.

Un *problema de decisión* es aquél en el que hace una pregunta y se espera una respuesta de "sí" o "no". Un *problema NP-completo* es aquél para el cual no se sabe si hay un algoritmo que en tiempo polinomial lo resuelva, pero las respuestas se pueden verificar en dicho tiempo (ver capítulo 5 para una introducción más detallada). Por ejemplo, si tenemos un conjunto de números enteros y queremos saber si hay un subconjunto no vacío tal que la suma de sus elementos dé como resultado cero, es NP-completo. Es decir, no existe algoritmo que lo resuelva rápidamente, pero sí se pueden verificar las respuestas rápido. Como ejemplo, supongamos que tenemos el conjunto $\{-7, -3, -2, 5, 8\}$. Con el subconjunto $\{-3, -2, 5\}$ la respuesta es sí, y es muy rápido verificarlo. Pero encontrar este subconjunto es muy lento en general. Este problema es el de la suma del subconjunto (Subset sum problem). De hecho el problema de la partición, cuya reducción veremos en el capítulo 5, es un caso especial de este último problema que vimos como ejemplo.

En general el enfoque que le damos al problema es el de destruir todas las barreras con el menor número de disparos. Este enfoque puede verse como si estuviéramos a la ofensiva. Sin embargo, utilizando estos mismos algoritmos podríamos acomodar las barreras de tal manera que maximizamos el número de disparos necesarios para destruir, convirtiendo el problema de modo que estamos a la defensiva. Es decir, que maximizamos el número de disparos para evitar que roben el tesoro.

A continuación veremos diversas aplicaciones del problema. Por ejemplo, supongamos que se quiere estudiar alguna capa interna del planeta. Primero tenemos que llegar a ella perforando. En [11] y [12] se presenta un proyecto en el cual se quiere atravesar la corteza terreste del planeta, y llegar al manto para obtener muestras por primera vez (figura 1.7). Dicen que estudiar estas

muestras sería el equivalente a estudiar rocas lunares, que son igual de difíciles de obtener. Los estudios de estas muestras podrían ayudar a saber más sobre terremotos y los orígenes de la Tierra.

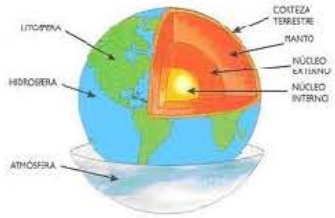

Figura 1.7: Capas de la Tierra.

Este problema puede ser modelado con el nuestro si vemos los componentes del suelo que conforman una capa como si fueran las barreras, y los taladros como si fueran los disparos. Nuestro objetivo sería destruir las barreras para abrirnos paso, pero minimizando el uso de los taladros.

Otra aplicación del problema puede verse como el juego "Angry Birds" (figura 1.8). En este juego tenemos unas aves que se lanzan para destruir a unos cochinitos. Estos cochinitos se encuentran protegidos por diferentes objetos, como pueden ser fortalezas de madera, de cristal, de piedra, entre otros. El objetivo es destruir a todos los cerdos pero utilizando la menor cantidad de aves (disparos). Existen diferencias con nuestro problema, como por ejemplo la física de los disparos y los objetos que protegen a los cerdos; también se diferencian en que los disparos de nuestro problema se originan desde arriba. Se parecería más a nuestro problema si dejáramos caer a las aves desde arriba, y si todas destruyeran las barreras que tocan. Las aves rojas grandes destruyen todo a su paso en el juego, así que para nuestro problema necesitaríamos una que tenga esta propiedad y que además se mueva sólo en línea recta.

Figura 1.8: Juego de Angry Birds.

Ahora imaginemos que quisiéramos destruir tabiques de hielo como alguien que practica karate (figura 1.9). Este problema quedaría modelado con nuestro problema de la siguiente manera. Los disparos serían los golpes y podemos darles diferente potencia a cada uno. Entonces sería como el caso en el que tenemos un conjunto finito de disparos (suponemos que el practicante de karate se cansa) de diferentes potencias (el karateca puede golpear con diferente fuerza).

Figura 1.9: Karateca rompiendo hielo.

1.1. Trabajo relacionado

Ninguno de los problemas presentados en este trabajo se han abordado con anterioridad, especialmente utilizando disparos con diferentes potencias. A continuación veremos trabajos relacionados y los problemas que atacan, con el objetivo de ver las similitudes con este trabajo.

Un primer artículo relacionado con el problema estudiado en este trabajo es el presentado en [7]. La motivación de los autores de dicho trabajo fue por el creciente interés en los ochentas por las gráficas por computadora. El problema que tratan es aquél en el que se tiene un conjunto de segmentos de línea en el plano y se quiere saber si existe una línea, llamada *línea punzante* (stabbing line), la cual se intersecta con todos los elementos de dicho conjunto. Hablando en términos de visibilidad, este problema puede ser interpretado como encontrar una dirección desde la cual uno puede mirar a través de un conjunto de puertas, donde una puerta es un intervalo. Los autores afirman que parece no haber un algoritmo trivial de fuerza bruta para dicho problema. En [8] se presenta un algoritmo que resuelve este problema en $O(n^2 \log n)$ para encontrar la línea punzante para n segmentos de línea. Sin embargo, en [7] se mejora la complejidad a $O(n \log n)$, y además se calcula la descripción de todas las líneas punzantes. Desde esta descripción, se puede encontrar una línea punzante específica en tiempo constante (si existe tal línea). Además, la descripción permite saber si ciertas líneas específicas son líneas punzantes.

Los artículos mencionados a continuación tienen cierta similitud con nuestro problema y sus variaciones. Sin embargo, hay que notar que los problemas que abordan ellos tienen intervalos de línea con rotación arbitraria en el plano, mientras que nosotros vemos nuestros intervalos de manera apilada; es decir, como si viéramos un muro de tabiques de frente en vez que desde arriba. El siguiente problema (y sus variantes) es uno parecido al que se estudia en este trabajo. El primer artículo propone un problema y los siguientes toman el mismo problema para mejorar la complejidad de los algoritmos. De la misma forma, también proponen variantes y sus respectivos algoritmos.

En [3] se plantea que se quiere destruir segmentos ajenos en el plano, pero éstos tienen orientación arbitraria. Los disparos se originan a partir de una recta cualquiera del plano y pueden tener cualquier dirección. El problema consiste en encontrar un punto p sobre la recta tal que al disparar desde él se destruyan todos los intervalos con la menor cantidad de disparos. El algoritmo para resolver este problema tiene una complejidad de $O(n^3)$. Este problema tiene el

nombre del "problema de localización del tirador" (Shooter location problem).

En [4] se toma el mismo problema de [3] pero en vez de restringir a una recta el punto de disparo, éste se busca en todo el plano. Utilizando el plano dual se desarrolla un algoritmo de complejidad $O(n^5 \log n)$ que resuelve el problema. Adicionalmente se incluye otro algoritmo de complejidad $O(n^5)$ cuyo factor de aproximación es dos.

Siguieron otros autores que tomaron el mismo problema y trataron de disminuir la complejidad, como [5]. Lograron un algoritmo de complejidad $O(n^4 + nK)$, donde K es un número que definen como de celdas domintantes. Dicen que el peor caso de K puede ser $O(n^4)$ pero que en la práctica es menor. El último restultado de este problema está en [6], cuya complejidad es de $O(n^4 \log n)$. Este algoritmo revisa exhaustivamente todas las celdas generadas por los intervalos de línea ($O(n^4)$), mantiene una cobertura mínima de clanes y un conjunto independiente máximo de una gráfica arco-circular ($O(\log n)$ amortizado).

Una gráfica de intervalos es una gráfica tal que cada vértice representa un intervalo de una recta ℓ. Dos vértices son adyacentes si sus intervalos correspondientes se intersectan. En [2] muestran cómo encontrar, para una familia de intervalos, un conjunto independiente máximo, una cobertura mínima por conjuntos disjuntos completamente conectados (clanes), y un clan máximo, en $O(n \log n)$ ($O(n)$ si los extremos de los intervalos están ordenados). También muestran cómo encontrar en $O(n^2)$ un conjunto independiente máximo y una cobertura mínima por conjuntos disjuntos completamente conectados en gráficas arco-circulares más generales. En particular nos servirá el algoritmo para encontrar un conjunto independiente máximo. Para más detalles sobre las definiciones de conceptos que acaban de mencionar, ver el Capítulo 2.

En [9] se presenta un problema que es el más parecido al estudiado en el presente trabajo. El problema (y variantes) que estudian es el de cubrir segmentos de línea, rayos, y líneas en el plano ortogonales. Demuestran que los problemas con conjuntos segmentos de línea horizontales/verticales son NP-completos, y muestran que variantes que involucran rayos o líneas paraleleas se pueden resolver en tiempo polinomial. Los problemas estudiados en dicho artículo son los siguientes:

1. Problema del conjunto dominante de segmento ortogonal. Lo que se busca es encontrar un subconjunto de cardinalidad mínima tal que cada segmento sea intersectado por otro segmento del subconjunto.
2. Problema de cobertura de segmento ortogonal. Lo que se quiere es encontrar un subconjunto de cardinalidad mínima segmentos verticales tal que todos los segmentos horizontales sean intersectados por este subconjunto. Además se estudian variantes con rayos en vez de segmentos:
   - En el problema de intersectar segmentos con rayos, el conjunto de líneas verticales ahora se toma como un conjunto de rayos. Lo que se busca es encontrar el subconjunto con la mínima cardinalidad tal que los rayos verticales se intersecten con todos los segmentos horizontales.

- El problema de intersectar rayos con segmentos es el siguiente: encontrar un subconjunto de cardinalidad mínima de segmentos verticales que se intersecten con todos todos los rayos horizontales.
- La última variante es la de intersectar rayos con rayos: se quiere encontrar un subconjunto de cardinalidad mínima de rayos verticales que se intersecten con todos todos los rayos horizontales.

En dicho artículo prueban que algunas variantes son problemas NP-completos, mientras que otros casos especiales se pueden resolver en tiempo polinomial usando programación dinámica. Ciertas variantes son instancias de problemas de cobertura de conjuntos (set cover problems), y por esto existen algoritmos de aproximación de complejidad $O(\log n)$ basados en una heurística ávida (greedy). El siguiente cuadro de resultados (figura 1.10) es presentado en ese artículo. Es de especial interés para nosotros pues estas variantes son parecidas a las estudiandas en el presente trabajo. La línea sin flechas representa segmentos de línea (verticales/horizontales), la línea con una sola flecha representa un rayo junto con su dirección, mientras que la línea con dos flechas representa líneas infinitas.

	$\mid$	$\downarrow$	$\updownarrow$
—	NP-complete	$O(n^3)$	$O(n \log n)$
$\rightarrow$	$O(n^6)$ $O(n \log n)$ 2-approx	$O(n \log n)$	$O(n)$
$\leftrightarrow$	$O(n \log n)$	$O(n)$	$O(1)$

Figura 1.10: Resumen de resultados para el problema de cubrir segmentos ortogonales.

De las variantes que se estudian en este artículo, el de intersectar segmentos horizontales con rayos verticales (Stabbing segments with rays: SSR) es el más parecido al problema estudiado en este trabajo. La diferencia principal radica en que en este artículo el conjunto de rayos verticales ya está dado, mientras que en el que estudiamos nosotros, queremos asignar los rayos de tal modo que formemos el conjunto de cardinalidad mínima que intersecta con todos los segmentos horizontales. En general este problema es el más parecido al que estudiaremos en el presente trabajo.

El problema SRR de ellos es resuelto con programación dinámica. Lo que hacen es dividir el problema en diferentes secciones, las cuales son formadas con el conjunto de rayos verticales que ya están ordenados con respecto a su coordenada x. Dos rayos verticales especiales delimitan todo el plano: el rayo r_0 es el de más a la izquierda, mientras que el rayo r_{n+1} es el del extremo derecho. En cada sección se va resolviendo el subproblema de encontrar un subcojnuto de rayos verticales tal que todos los segmentos horizontales de dicha sección sean atravesados por al menos un rayo vertical (figura 1.11). El objetivo total es calcular un subconjunto que cubra todos los segmentos horizontales.

Usando recursión se resuelven los subproblemas, llegando así a la resolución total del problema. Se tienen m segmentos horizontales, entonces encontrar el segmento con la coordenada y máxima toma $O(m)$. Existen n rayos, por lo que elegir el k-ésimo rayo que separa una sección toma $O(n)$, y calcular dentro

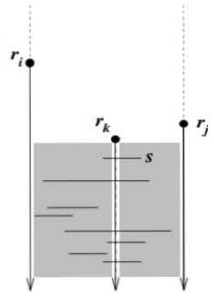

Figura 1.11: Definición del subproblema para el problema SSR.

de cada sección toma $O(n^2)$. Por tanto, el algoritmo que presentan tiene una complejidad de $O(n^2(m+n))$ en tiempo, y $O(n^2+m)$ en espacio.

Es natural que nuestros algoritmos y complejidades sean diferentes a los que presentaremos pues son problemas diferentes, comenzando por el hecho de que en nuestro problema tenemos la libertad de elegir dónde colocar los rayos verticales, mientras que en el problema SSR los rayos ya están dados. Debido a esto, ellos hacen uso de programación dinámica, mientras que nosotros utilizamos un algoritmo ávido que va "creando" rayos verticales donde parece ser (y lo es) el mejor lugar. El capítulo 3 es el dedicado a la variante más similar al problema SSR y dónde viene dicho algoritmo ávido, por lo que las diferencias se notarán más en dicho capítulo.

Capítulo 2

Teoría de Gráficas

Muchas veces nos encontramos con situaciones en las que afrontar un problema tal cual como nos es presentado es difícil. Por ejemplo, dibujar una imágen en el monitor con una computadora puede parecer difícil la primera vez que vemos este problema. Sin embargo, si a las imágenes las consideramos como conjuntos de pixeles, entonces podemos manipularlas más libremente. Lo que estamos haciendo es tomar un objeto y quedarnos sólo con lo que nos interesa de él. Esto es hacer una abstracción. Una gran ventaja de hacer una abstracción es tomar algo y verlo más como otra cosa que conocemos.

Nuestro problema original de destruir una configuración de barreras puede representarse de diferentes formas. Lo que haremos será representarlo como una gráfica y tomar algoritmos y propiedades ya conocidos para poder generar nosotros más algoritmos pero teniendo en cuenta las características de interés del problema original. Además, este capítulo nos ayudará a establecer una notación y diferentes definiciones, lo cual será útil para las demostraciones.

2.1. Definiciones básicas

Recurrimos a la noción de gráfica para poder abstraer una relación entre las barreras, de modo que nos será más fácil ver al conjunto de barreras como una gráfica. Una gráfica G es un par $G = (V,E)$, donde V es un conjunto finito de $n = |V|$ elementos llamados *vértices* y $E \subseteq \{\{x, y\}|x, y \in V, x \neq y\}$ es un conjunto de $e = |E|$ pares de vértices no ordenados llamados *aristas* (figura 2.1).

Para un par vértices diferentes x y y, decimos que x es *adyacente* a y, (o de igual forma, y es adyacente a x) si $\{x, y\} \in E$; de lo contrario, decimos que son *independientes*. Es decir, si dos vértices están unidos por una arista decimos que son adyacentes; si no existe tal arista decimos que son independientes (figura 2.2).

Un conjunto $V' \subseteq V$ de vértices es llamando *conjunto independiente* si todos los elementos de V' son independientes entre sí. En la figura 2.3 el conjunto de vértices azules es independiente.

Se denomina conjunto independiente *maximal* a un conjunto independiente que no es subconjunto propio de otro conjunto independiente. Es decir, es un

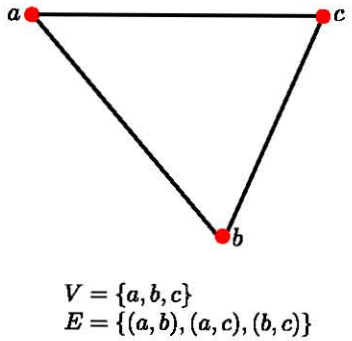

Figura 2.1: Gráfica

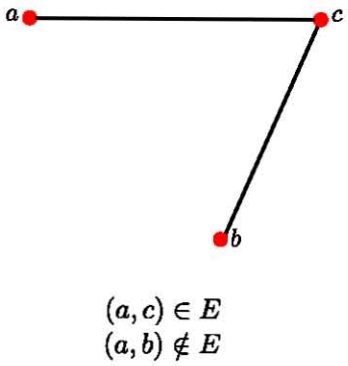

Figura 2.2: Vértices independientes y vértices adyacentes.
a es adyacente a c, pero independiente a b

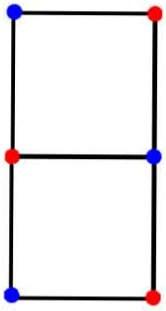

Figura 2.3: Conjunto independiente (vértices azules).

conjunto tal que no se puede agregar otro elemento al conjunto sin violar la independencia. Se dice que un conjunto independiente es *máximo* es aquél con un número máximo de vértices entre todos los conjuntos independientes de una gráfica. Entonces, un conjunto independiente máximo es maximal, pero no viceversa (figura 2.4). El número de vértices en un conjunto independiente máximo

será denotado por $\alpha(G)$, y es un invariante de G.

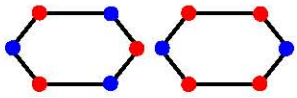

Figura 2.4: Conjuntos independientes (vértices azules)
El conjunto de vértices azules en la figura izquierda es máximo, mientras que el de la derecha es maximal.

En particular, encontrar el conjunto independiente máximo nos dará gran ventaja para poder destruir óptimamente las barreras adyacentes (barreras que se traslapan o empalman).

Un conjunto $V' \subseteq V$ de vértices es llamado conjunto *completamente conectado* si todos los vértices en V' son adyacentes unos con otros (figura 2.5). Podemos ver a este conjunto como una gráfica completa. Una gráfica completa es aquella en la que cada par de vértices está unido por una arista (todo vértice es adyacente a todos los demás).

Un *clan* (clique) es un conjunto completamente conectado *maximal*, es decir, $V' \subseteq V$ es un clan si V' es un conjunto completamente conectado y no hay otro conjunto completamente conectado V'' tal que $V'' \supset V'$.

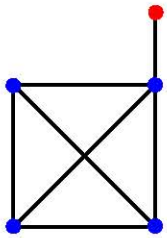

Figura 2.5: Conjunto completamente conectado dentro de una gráfica.
El conjunto de vértices azules está completamente conectado.

Un clan es *máximo* cuando tiene un número máximo de vértices entre todos los clanes. El número de vértices en un clan máximo será denotado por $\beta(G)$.

Una *gráfica de intersección* es una gráfica que representa el patrón de intersecciones de una familia de conjuntos. Formalmente, una gráfica de intersección se define como una gráfica no dirigida $G = (V, E)$ formada por una familia de conjuntos S_i, donde $i = 0, 1, 2, \ldots, n$, en la cual se crea un vértice v_i por cada conjunto S_i, conectando dos vértices v_i y v_j por una arista si los dos conjuntos tienen intersección no vacía, es decir, $E = \{\{v_i, v_j\} | S_i \cap S_j \neq \emptyset\}$. Por ejemplo, se puede pensar en un grupo de gente a las cuales les gusta un género musical, como el rock. A otro grupo de gente les gusta el pop. Además hay personas a las cuáles les gustan ambos géneros. Si se representa esta relación en una gráfica, se tendría un vértice para el grupo de gente a la cual le gusta el rock, otro vértice para la gente del pop, y una arista que une a ambos vértices porque hay gente a las que les gustan ambos (figura 2.6).

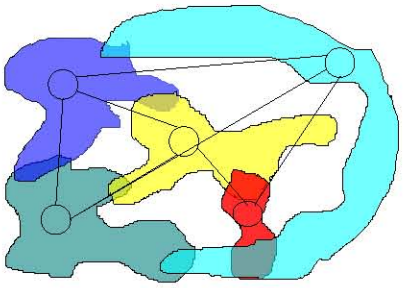

Figura 2.6: Gráfica de intersección
Los conjuntos son vértices, y las intersecciones son aristas.

La gráfica de intersección nos puede ayudar a representar algo que originalmente no está pensado como una gráfica, como veremos a continuación.

Una *gráfica de intervalos* es una gráfica de intersección de intervalos. Tiene un vértice por cada intervalo en el conjunto, y una arista entre cada par de vértices correspondiente a los intervalos que se intersectan. Es decir, sea $\{L_1, L_2 \dots L_n\}$ un conjunto de intervalos. La gráfica de intervalos correspondiente es $G = (V, E)$ donde:

- $V = \{L_1, L_2 \dots L_n\}$ y
- $\{L_a, L_b\} \in E \leftrightarrow L_a \cap L_b \neq \emptyset$

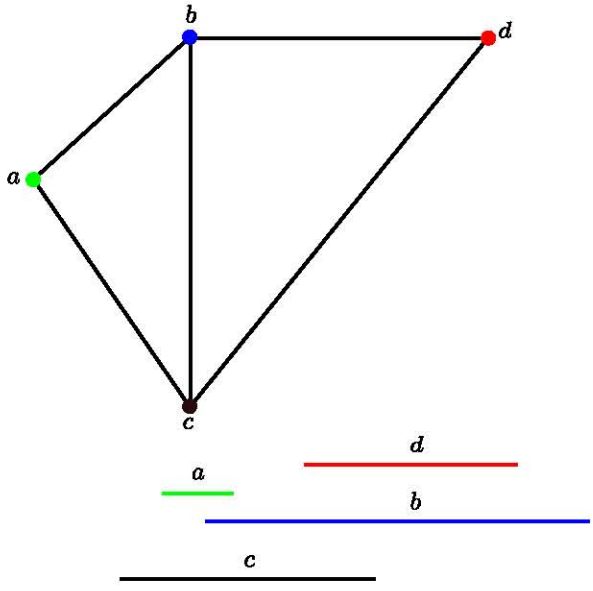

Figura 2.7: Gráfica de intervalos

La gráfica de intervalos nos ayudará a representar de otra forma la configuración de barreras, de modo que podemos aprovechar diferentes propiedades de las gráficas, tales como las previamente mencionadas (clanes y conjuntos independientes, entre otros).

Si el conjunto de intervalos es un conjunto de arcos sobre un círculo, la gráfica G es llamada una *gráfica arco-circular*. En [2] se establece que la clase de gráficas de intervalos está propiamente contenida en la clase de las gráficas arco-circulares.

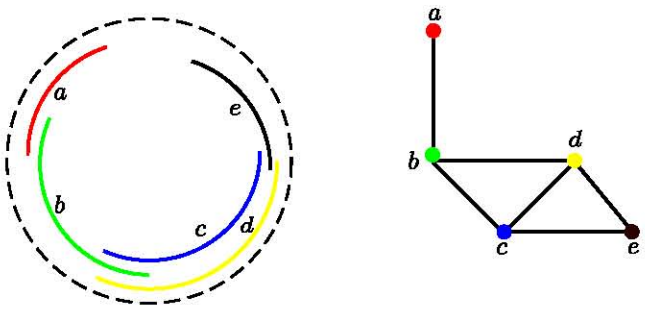

Figura 2.8: Gráfica arco-circular

Además, en [2] se menciona que cada gráfica de intervalos es una gráfica arco-circular ya que podemos representar los intervalos como arcos en un círculo, pero que existen gráficas arco-circulares que no son gráficas de intervalos. Lo más que podemos hacer para "convertir" una gráfica arco-circular en una gráfica de intervalos es hacer un corte en la arco-circular, y después "estirarla" de modo que los arcos los hacemos rectas. La gráfica resultante es una gráfica de intervalos, pero no es isomorfa a la arco-circular original porque si no existe punto de la gráfica arco-circular donde podamos cortar radialmente sin partir un arco, los intervalos partidos resultantes no estarán en correspondencia uno a uno con los originales. Cada intervalo original cortado dará lugar a dos nuevos segmentos (figura 2.10).

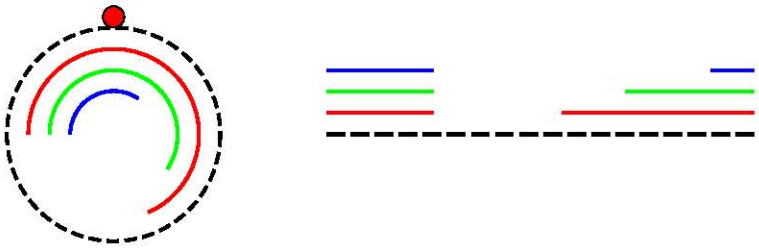

Figura 2.9: Gráfica arco-circular cortada donde hay intervalos.

El algoritmo descrito en [2] para encontrar un conjunto independiente máximo en $O(n \log n)$ es el siguiente.

Figura 2.10: Gráfica arco-circular cortada donde no hay intervalos.

Algoritmo 1 Obtención de un conjunto independiente máximo.

Entrada: Los $2n$ extremos de las barreras.
Salida: Obtención de un conjunto independiente máximo MIS.

```
1:  Ordena los 2n puntos con respecto a su coordenada x
2:  Sea p el punto más a la izquierda
3:  Sea MIS = ∅
4:  mientras haya barreras sin borrar hacer
5:    si p es un extremo izquierdo entonces
6:      p toma el valor del siguiente punto
7:    si no
8:      Agrega la barrera que tiene a p como extremo derecho a MIS
9:      Borra todas las barreras que se intersectan con p
10:     si hay barreras sin borrar entonces
11:       p se vuelve punto más a la izquierda
12:     fin si
13:   fin si
14: fin mientras
```

Este algoritmo va recorriendo los $2n$ puntos que son los extremos de las barreras, por lo que en el peor caso recorre todos, originando una complejidad de $O(n)$. Sin embargo, esto es suponiendo que los $2n$ puntos están ordenados, ya que si no lo están entonces hay que ordenarlos, lo cual nos eleva la complejidad a $O(n \log n)$.

De hecho veremos que este algoritmo es muy similar al algoritmo 2 porque ambos encuentran un conjunto independiente máximo. Sólo que el segundo lo hace como un "efecto colateral" mientras busca las mejores posiciones para disparar.

A continuación ilustramos una ejecución del algoritmo 1.

Supongamos que tenemos la configuración de la figura 2.11. El algoritmo primero ordena los extremos de las barreras con respecto a su coordenada x, de modo que "el primer" extremo será el que esté más a la izquierda; "el segundo" el que se encuentre inmediatamente a la derecha del primero, y así sucesivamente.

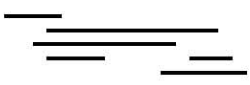

Figura 2.11: Encontrando un conjunto independiente máximo.

El algoritmo se fija en qué tipo de extremo es el punto que está visitando. Es decir, se fija si es un extremo izquierdo o un extremo derecho. El primer punto será un extremo izquierdo, por lo que el paso 5 se aplica y cambiamos de punto. El algoritmo seguirá cambiando el punto actual mientras no encuentre un extremo derecho. Cuando lo encuentre, agregará al conjunto independiente máximo a la barrera que tenga a dicho punto como extremo. Las barreras adyacentes a este punto serán removidas de la configuración pues ya están "cubiertas" por la barrera que acabamos de agregar. En la figura 2.12 el primer extremo derecho que encontramos es el de la barrera a (color verde), y la línea roja vertical nos muestra qué barreras dejaremos de considerar para el conjunto independiente máximo.

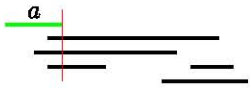

Figura 2.12: Encontrando un conjunto independiente máximo (continúa).

Al seguir con la ejecución, el algoritmo ya no considerará las barreras adyacentes a a, por lo que encontrará el extremo derecho de c (barrera azul). Ahí agregará a c al conjunto independiente máximo y quitará la barrera que intersecta con c (figura 2.13). Como ya no tenemos barreras el algoritmo termina. Dando como resultado el conjunto independiente máximo $\{a, c\}$. Hay que notar que puede existir más de un conjunto independiente máximo. Sin embargo, veremos más adelante que disparar con potencia infinita en los extremos derechos de las barreras que encuentre implícitamente el algoritmo 2 será lo óptimo.

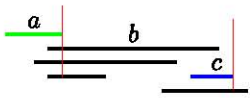

Figura 2.13: (Continuación) Encontrando un conjunto independiente máximo.

Capítulo 3

Disparos de potencia infinita

En este capítulo abordaremos el caso más trivial e irreal en general. Supondremos que tenemos disparos con potencia infinita, es decir, un disparo rompe todas las barreras que atraviesa. Parece no tener una aplicación real porque no podríamos encontrar una potencia infinita. Sin embargo, si de antemano sabemos que todos nuestros disparos tienen potencia mayor o igual al número de barreras, las propiedades y algoritmos que veremos aquí se siguen manteniendo.

Parece bastante simple el problema a primera vista, ya que lo único que necesitamos es disparar donde esté la mayor concentración de barreras. Y de hecho en ejemplos pequeños lo es. Sin embargo si tenemos una configuración con más de un millón de barreras, puede que no sea tan fácil ver dónde está la mayor concentración de barreras. Si las barreras están distribuidas más o menos uniformemente, veremos sólo un mar de barreras.

Necesitamos poner especial atención en los extremos de los intervalos (barreras). Podemos considerar un extremo como el principio, y el otro extremo como el fin. Al fijarnos en estos extremos podemos aprovechar la longitud de las barreras. Más adelante veremos la importancia de estos extremos.

A continuación introduciremos notación que utilizaremos en lo restante de este trabajo.

Un segmento de línea horizontal i será representado por sus dos puntos extremos p y q en orden de izquierda a derecha, respectivamente. p será llamado *cola* de i, o $t(i)$, y q será llamado la *cabeza* de i, o $h(i)$ (figura 3.1).

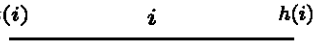

Figura 3.1: Una barrera y sus extremos.

Al número de disparos para destruir todas las barreras lo denotaremos por d.

Aprovechando la potencia infinita, queremos buscar aquellas barreras que se intersecten con la mayor cantidad de barreras porque al disparar en la intersección destruiremos más con menos disparos. Un primer algoritmo podría seguir esta dinámica. La idea sería buscar la barrera con el extremo con la mayor

cantidad de intersecciones. Después disparar en ese extremo, destruir todas las barreras que intersecten, e iterar hasta que no haya barreras. A continuación veremos una ejecución del algoritmo que sigue esta idea:

El algoritmo primero recibe los $2n$ extremos de las n barreras de la configuración (figura 3.2).

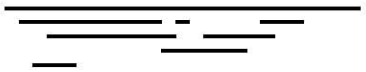

Figura 3.2: Una configuración original

Se ordenan primero los puntos para hacer un barrido horizontal ($O(n \log n)$). Después se hace el barrido de la siguiente manera: se posiciona en el extremo más a la izquierda, se cuentan las barreras con las que se intersecta este punto (figura 3.3), y se pasa al siguiente punto (extremo siguiente). Esto se hace con todos los puntos (figura 3.4). Sin pérdida de generalidad, el barrido se hace de izquierda a derecha.

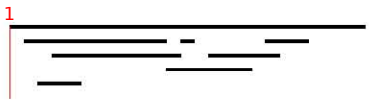

Figura 3.3: Contando las intersecciones en el primer punto

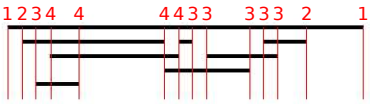

Figura 3.4: Contando las intersecciones en todos los puntos

Luego se posiciona en el punto con el contador más alto de intersecciones (eligiendo arbitrariamente en caso de empate), se dispara, y se destruyen las barreras que intersecta el disparo, incluyendo la barrera que tiene al punto con el contador. Esto concluye una iteración. Si hay más barreras, se vuelve a iterar (figura 3.5) y termina cuando no haya más barreras.

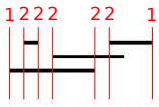

Figura 3.5: Después del primer disparo, si hay más barreras se vuelve a contar y se sigue iterando

A pesar de que este algoritmo destruye todas las barreras con la menor cantidad de disparos, contar las intersecciones es muy lento. Este algoritmo sería bueno si de antemano supiéramos que cada configuración tendrá muchísimas intersecciones, porque de esa manera cada disparo destruirá muchas barreras,

y habría que analizar mucho menos barreras en cada iteración. Sin embargo, la complejidad es de $O(n^2)$ porque el peor caso sería tener una configuración donde no hay intersecciones (figura 3.6). En total son n barreras, por lo que se tienen que realizar n iteraciones (una para cada barrera), y en cada una de éstas comparar la i-ésima barrera con n barreras, por lo que el tiempo total es de $O(n \times n)$, es decir, $O(n^2)$.

Figura 3.6: Un disparo para cada barrera.

En este capítulo veremos un algoritmo que en general es más simple, rápido, y resuelve óptimamente el problema. El algoritmo consiste en hacer un barrido de línea de un extremo a otro poniendo sólo atención a la *cabeza h(l)* de cada barrera *l*. Por barrido de línea nos referimos a que se visita cada extremo de cada intervalo en orden de izquierda a derecha. Imaginemos que tenemos una línea vertical posicionada en el punto más a la izquierda. Esta línea se moverá hacia la derecha e irá tocando los extremos de cada intervalo, en los cuales se detendrá a verificar qué tipo de extremo es. Se disparará en cada *h(l)* que se encuentre.

Algoritmo 2 Minimización de disparos con potencia infinita.

Entrada: Los $2n$ extremos de las barreras.
Salida: Un conjunto M de verticales que cubre la configuración.

```
1:  Sea M = ∅ el conjunto de verticales que cubrirá la configuración
2:  Ordena los 2n puntos con respecto a su coordenada x
3:  Sea p el punto más a la izquierda
4:  mientras haya barreras sin destruir hacer
5:    si p es una cola entonces
6:      p toma el valor del siguiente punto
7:    si no
8:      Agrega la vertical de p a M
9:      Dispara en p destruyendo todas las barreras que se intersecten con la
        vertical que pasa por p
10:     si hay barreras sin destruir entonces
11:       p se vuelve punto más a la izquierda
12:     fin si
13:   fin si
14: fin mientras
15: devolver M
```

A continuación veremos una ejecución del algoritmo 2 y después analizaremos sus propiedades.

Se comienza por recibir los $2n$ extremos de las barreras (figura 3.7).

Después se ordenan estos puntos, lo cual toma $O(n \log n)$. Una vez terminado esto, el algoritmo sabe cuál es el punto más a la izquierda (figura 3.8), el que

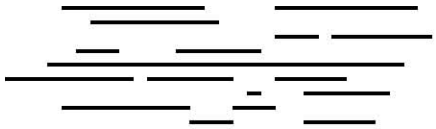

Figura 3.7: Configuración de barreras inicial.

le sigue a la derecha, y así sucesivamente.

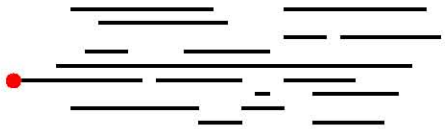

Figura 3.8: El algoritmo se posiciona en al punto más a la izquierda.

Lo anterior es importante porque ya se puede hacer el barrido de izquierda a derecha. De nuevo, un barrido consiste en ir visitando cada punto de izquierda a derecha en orden de aparición. Al visitar un punto se revisa qué tipo de extremo es, es decir, si es una cola o una cabeza. Como lo único que importa encontrar son cabezas, se ignoran las colas y se pasa al siguiente punto para revisarlo. Una vez que se encuentra una cabeza (figura 3.9), se dispara eliminando a la barrera que tiene a este punto como cabeza, y a todas las barreras que se intersecten con este punto (figura 3.10).

Figura 3.9: El algoritmo busca la primera cabeza que queda en la configuración.

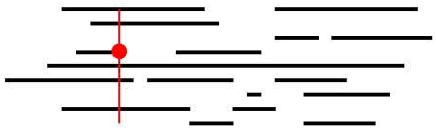

Figura 3.10: El algoritmo destruye todas las barreras que se intersectan con la cabeza encontrada.

La complejidad de este algoritmo es menor que el anterior porque únicamente se ordenan los puntos y se buscan aquellos que son cabezas de barreras. El problema con el anterior es que se tienen que calcular las intersecciones en cada iteración, y a diferencia de eso, en este algoritmo no interesa saber esto pues se aprovecha la potencia infinita y con seguridad se sabe que se destruirán todas las barreras que el disparo atraviese.

A continuación se analizarán las propiedades del algoritmo.

Es fácil ver que con un solo barrido se destruirán todas las barreras porque conforme se avanza con el barrido no van quedando barreras. Se puede probar por contradicción: Supongamos que el algoritmo acaba de aplicar el paso 8 y está a punto de disparar en la cabeza de la i-ésima barrera en el paso 9. Ahora supongamos que hay una cabeza $h(i-1)$ de una barrera $i-1$ a la izquierda de i. El paso 2 del algoritmo ya ordenó los puntos, por lo que no se puede saltar ninguno. A la izquierda de $h(i-1)$ sólo puede haber colas ya que el paso 9 destruye todas las cabezas en orden, por lo que tener todavía una cabeza ahí es una contradicción. Esta observación nos lleva a la **invariante** del algoritmo: Sea f_i con $\{i = 1, 2, 3, \ldots, d\}$ una vertical del conjunto M. Al final de la iteración k, todo intervalo de la configuración inicial que tiene una cola $t(i)$ con $t(i) \leq f_k$, está cubierto por M (figura 3.11).

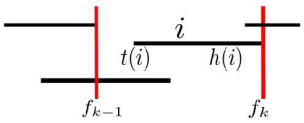

Figura 3.11: Invariante del algoritmo: todas las barreras a la izquierda de un disparo están cubiertas.

Lema 1. *En el algoritmo 2, la complejidad del ciclo (pasos 4 a 14) es $O(n)$.*

Demostración. La complejidad no puede ser mayor ya que en el peor de los casos, las barreras están de forma en que ninguna se intersecta con otra (figura 3.6). Y en los pasos 4 a 14 el algoritmo revisaría las n barreras (todas) y no más. □

Hasta este momento sabemos que el algoritmo destruye todas las barreras. Por sí mismo parece óptimo y por pura intuición podríamos decir que lo es. Sin embargo, daremos las pruebas necesarias. Primero demostraremos que el algoritmo encuentra un conjunto independiente máximo. Después, que al destruir tal conjunto independiente máximo que encuentra el algoritmo destruye todas las barreras. Finalmente, acotaremos el número de disparos: demostraremos que el número de disparos necesarios es por lo menos tan grande como un conjunto independiente máximo y, finalmente, que el número de disparos necesarios es exactamente igual al tamaño de dicho conjunto.

En [2] se describe un algoritmo similar (algoritmo 1) que va haciendo un barrido y revisando el extremo final de cada barrera (cabezas, en nuestro caso), y no incluyen demostraciones de las propiedades del algoritmo. Ahora demostraremos que nuestro algoritmo destruye todas las barreras y además encuentra un conjunto independiente máximo (esto último se puede ver como efecto colateral).

Teorema 1. *El conjunto M que construye el algoritmo 2 es independiente máximo.*

Demostración. Como está definido en el capítulo 2, un conjunto independiente es aquél en el que ninguno de sus elementos es adyacente a algún otro elemento

del conjunto. Es decir, en términos de nuestras barreras, es un conjunto compuesto por barreras que no se intersectan entre sí.
Recordemos que un conjunto independiente es máximo si tiene el mayor número posible de elementos. Nótese que puede existir más de un solo conjunto independiente máximo en una misma gráfica.

El algoritmo 2 dispara a la barrera i cuando encuentra a $h(i)$, eliminando las barreras adyacentes a i, de modo que va disparando sólo en barreras no adyacentes entre sí, formando así con las barreras similares a i un conjunto M de barreras que es un conjunto independiente, y como veremos a continuación, máximo. Nótese que un disparo no puede destruir a más de un elemento de M ya que es un conjunto independiente. Supongamos que M no es máximo. Entonces debería existir otro conjunto independiente mayor, pero para que este último existiera, por lo menos una barrera no sería adyacente a las barreras en las que se disparó, lo cual contradice la invariante del algoritmo de que toda barrera está cubierta por una vertical. □

Hay que hacer notar que disparando sólo en la cabeza de cada elemento de cualquier conjunto independiente máximo no siempre se destruyen todas las barreras. Por ejemplo, en la figura 3.12 el conjunto $\{b, c\}$ es un conjunto independiente máximo, pero disparar en sus cabezas no destruye todas las barreras.

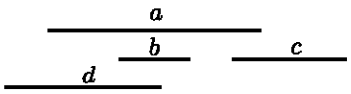

Figura 3.12: Una configuración con diferentes conjuntos independientes máximos.

Teorema 2. *Disparando sólo en la cabeza de cada elemento del conjunto independiente máximo que encuentra el algoritmo 2 se destruyen todas las barreras.*

Demostración. Supongamos que no es cierto. Entonces nuestro algoritmo encontraría un conjunto tal que al disparar en las cabezas de los elementos del conjunto, dejaría barreras sin destruir. Por ejemplo, en la figura 3.12 tenemos el conjunto independiente máximo $M = \{b, c\}$. Si se dispara ahí, nos quedaría d sin destruir. Sin embargo, el algoritmo hace un barrido de izquierda a derecha y dispara en cuanto encuentra la cabeza de una barrera que no ha sido destruida. Al disparar en esta barrera también se destruyen todas las barreras adyacentes a esta. Por la invariante sabemos que si se dispara en una barrera todas las barreras a la izquierda a partir de la cabeza donde dispara son destruidas, lo cual indica que conforme vamos barriendo, no se dejan barreras a la izquierda sin destruir. Esto nos lleva a una contradicción de que quedaron barreras. □

Teorema 3. *El número d de disparos necesarios para destruir todas las barreras es por lo menos tan grande como el tamaño de un conjunto independiente máximo.*

Demostración. Un conjunto independiente máximo es aquel que contiene la mayor cantidad de elementos no adyacentes, por lo tanto se necesitan por lo menos tantos disparos como elementos en el conjunto (uno para cada uno).

Supongamos que tenemos dos soluciones para una misma configuración: ambas supuestamente óptimas, pero una dice que se necesitan menos disparos que la otra. Esto significaría que una solución está identificando más barreras adyacentes que la otra, y que necesita menos disparos que la otra. Por tanto están identificando conjuntos independientes máximos diferentes. Pero si son soluciones con diferente cantidad de disparos para una misma configuración, tenemos una contradicción. □

Corolario 1 (Número de disparos exacto)**.** *El número de disparos necesarios d para destruir todas las barreras es exactmente tan grande como el tamaño de un conjunto independiente máximo.*

Demostración. Supongamos que tenemos una configuración en la cual existe un conjunto independiente máximo M y que en esta configuración existen dos barreras a y b que intersectan entre sí (figura 3.13). Ahora supongamos que tenemos una solución d que toma más disparos que el tamaño de M (en la figura 3.13 $|M| = 3$. Supongamos que $d = 4$). Esto nos lleva a que por lo menos dos barreras no intersectan, digamos que estas barreras son a y b. Pero si estas barreras no intersectaban, entonces había un conjunto independiente máximo M' mayor (en la figura 3.13, $|M'| = 4$ si a y b no intersectaran), llegando así a una contradicción. □

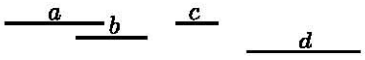

Figura 3.13: Conjunto independiente máximo de tamaño tres.

Entonces podemos concluir que nuestro algoritmo es óptimo:

Teorema 4 (Potencia infinita)**.** *El algoritmo 2 regresa el menor número de disparos necesarios para destruir todas las barreras en $O(n \log n)$.*

Demostración. Por las pruebas anteriores, tenemos lo siguiente: Primero se ordenan los extremos de las barreras, lo cual toma $O(n \log n)$. El algoritmo encuentra un conjunto independiente máximo M en $O(n)$, que es donde dispara. Conforme se va barriendo, se van destruyendo las barreras y no se va dejando ninguna detrás. El número de disparos es exactamente tan grande como el tamaño de un conjunto independiente máximo, en específico el que encontró el algoritmo, y como ya demostramos es el menor número posible. Por tanto, nuestro algoritmo es óptimo. □

Como tenemos potencia infinita, no nos importa si alguna barrera corta está atrapada entre muchas largas, ya que con un solo disparo podemos llegar a ella (figura 3.14).

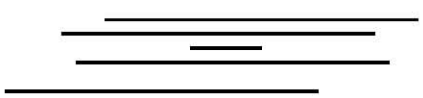

Figura 3.14: Barrera encerrada.

Algo curioso es que el orden en que se dispara puede cambiar si se barre en el otro sentido, es decir, de derecha a izquierda. Por ejemplo, en la figura 3.12, barriendo hacia la izquierda se dispararía en d y luego en c. Pero barriendo en el otro sentido, primero se dispararía en c y luego en b. A continuación demostraremos que el número de disparos será el mismo.

Nuestro algoritmo puede ser modificado ligeramente para construir una gráfica de intervalos de la siguiente forma:

Algoritmo 3 Construcción de una gráfica de intervalos a partir de una configuración de barreras.

Entrada: Los $2n$ extremos de las barreras.
Salida: Una gráfica de intervalos correspondiente a la configuración de barreras.

```
1: Ordena los 2n puntos con respecto a su coordenada x
2: Sea p el punto más a la izquierda
3: mientras haya cabezas sin revisar hacer
4:    si p es una cola entonces
5:       Crea un vértice en la gráfica con la misma etiqueta que la barrera
6:    si no
7:       Agrega una arista en la gráfica para cada una de las barreras que se
         intersectan con p
8:    fin si
9:    p toma el valor del siguiente punto
10: fin mientras
```

A continuación veremos una ejecución del algoritmo:

Primero se reciben los $2n$ puntos que son los extremos de las barreras y se ordenan (figura 3.15).

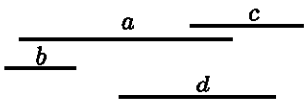

Figura 3.15: Configuración de barreras.

Como en los algoritmos previos, se hará un barrido de izquierda a derecha poniendo especial atención en qué tipo de punto es el que se visita; es decir, si es una cola o una cabeza. Si es una cola, se crea un vértice con la etiqueta de la barrera a la cual pertenece el punto. En la figura 3.16 el punto más a la izquierda (el primero) es la cola de b, por lo que se crea su vértice con su etiqueta.

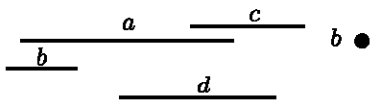

Figura 3.16: Configuración de barreras y el primer vértice de su gráfica de intervalos.

Si es una cabeza, se agregarán aristas en la gráfica para todas las barreras que se intersectan con este punto. Como ya se analizaron las colas de las barreras

con las que se intersecta la barrera que se está analizando, dichas barreras ya tienen vértices creados. Sin pérdida de generalidad, evitaremos el caso en el que las barreras comparten la misma coordenada x. En la figura 3.17 la primera cabeza que se encuentra es la de b. Antes la cola de a fue procesada, por lo que existe el vértice de a. Entonces al encontrar la cabeza de b, se agrega la arista en la gráfica la cual une los vértices de a y b.

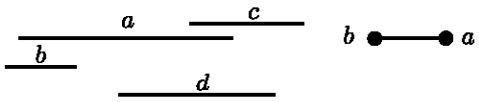

Figura 3.17: Configuración de barreras y la primera arista de su gráfica de intervalos.

La figura 3.18 muestra la gráfica de intervalos que se obtiene de la configuración inicial.

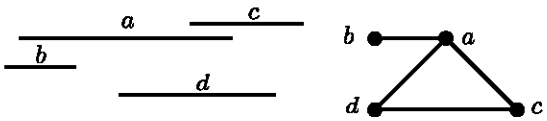

Figura 3.18: Configuración de barreras y su correspondiente gráfica de intervalos.

Claramente, este nuevo algoritmo hace lo mismo que el anterior con la diferencia de que en vez de destruir barreras adyacentes al disparar, crea aristas entre los vértices correspondientes a las barreras. A continuación demostramos que efectivamente construye una gráfica de intersección que representa la configuración de barreras.

Teorema 5. *El algoritmo 3 construye la gráfica de intersección de una configuración de barreras.*

Demostración. Supongamos que tenemos una configuración C y su respectiva gráfica de intersección G. Después corremos nuestro algoritmo y obtenemos una gráfica G' diferente a G. Si G' tiene vértices diferentes, significa que el algoritmo leyó alguna barrera de menos o de más, pero si leyó la misma configuración que originó a G, esto no puede ser, por construcción de código.
Si G' tiene un número diferente de aristas significaría que omitió o agregó aristas. Si tiene menos aristas, quiere decir que el algoritmo leyó que las barreras correspondientes a los vértices unidos por la arista faltante no son adyacentes. Sin embargo, ya probamos que el algoritmo 2 destruye las barreras adyacentes, entonces el algoritmo 3 en vez de destruir las barreras crea aristas. Lo cual nos lleva a que no puede ser posible que lea menos. Si G' tiene más aristas, entonces el algoritmo 3 encontró más barreras adyacentes que el algoritmo 2, lo cual es contradictorio pues están trabajando sobre la misma configuración. Si G' tiene el mismo número de aristas pero forma una relación diferente entre los vértices, el algoritmo estaría leyendo barreras no adyacentes como adyacentes y viceversa. Esto no puede ser porque recorre las barreras de la misma manera que el algoritmo anterior. □

Hay que hacer notar que sin importar el sentido del barrido, con el algoritmo anterior se obtiene la misma gráfica de intervalos, pero no siempre el mismo conjunto independiente máximo.

Teorema 6. *Si se cambia el sentido del barrido, el algoritmo 3 construye la gráfica de intervalos.*

Demostración. Supongamos que el algoritmo encuentra una gráfica G barriendo en un sentido, y otra gráfica G' diferente barriendo en el otro sentido. Los vértices no pueden cambiar pues están leyendo la misma configuración. Si las aristas fueran diferentes significaría que existe un número diferente de barreras adyacentes al leer en un sentido que en el otro, pero claramente esto no puede pasar pues la configuración permanece intacta. □

Observación 1. *El algoritmo 2 barriendo en un sentido encuentra un conjunto independiente máximo M del mismo tamaño que el otro M' encontrado barriendo en el otro sentido una misma configuración. En general M y M' serán diferentes porque las barreras tienen tamaños diferentes. Sin embargo, su tamaño es el mismo por lo siguiente. Supongamos que el tamaño es diferente. Esto quiere decir que en un sentido se leyeron unas barreras como adyacentes, y en el otro como si no lo fueran. Sin embargo esto no es posible pues la configuración no es modificada.*

Observación 2. *El algoritmo 2 hace el mismo número de disparos d sin importar el sentido del barrido. Por el corolario 1 tenemos que el número de disparos es tan grande como un conjunto independiente máximo.*

Con esto concluimos este capítulo el cual parece tener algoritmos bastante simples, pero toda la teoría que está detrás de ellos es fuerte. Estos algoritmos en general nos servirán si sabemos que la potencia de cada uno de nuestros disparos es mayor o igual que el total de barreras de la configuración. Por ejemplo, si tenemos sólo disparos de potencia cincuenta y tenemos sólo diez barreras, podemos utilizar el algoritmo 2 con confianza absoluta. Incluso si tuviéramos disparos con diferentes potencias pero todas mayores a diez, podemos de igual forma destruir dicha configuración con el algoritmo ya mencionado. De hecho en los capítulos siguientes abordaremos los casos en los que no todos los disparos tienen potencia por lo menos tan grande como el número de barreras que componen una configuración.

Capítulo 4

Disparos de potencia constante

Ahora estudiaremos un caso un poco más realista, donde nuestros disparos tienen potencia finita y la misma entre ellos. Es decir, tienen el mismo "calibre": un disparo de potencia dos destruye dos barreras si pasa por la vertical que las intersecta, por ejemplo. Esta potencia la denotaremos por k. Veremos primero un algoritmo ávido el cual hace una buena aproximación del óptimo. Después veremos otro par de algoritmos los cuales toman la configuración de barreras y la reacomodan (reconfiguran). Esta reconfiguración mueve únicamente las coordenadas y de las barreras, porque si podemos mover las coordenadas x, trivialmente podemos la configuración en forma de pirámide y disparamos, lo cual será siempre óptimo. Además, lo interesante es que si sólo movemos las coordenadas y, las gráficas de intervalos de las configuraciones quedan exactamente iguales pues las intersecciones de las barreras quedan intactas. Esto provoca que perdamos información de la gráfica y las propiedades de la teoría de gráficas no nos ayuden tanto. El algoritmo anterior de potencia infinita (algoritmo 2) no destruye todas las configuraciones porque ahora la potencia de los disparos está limitada.

Tomando nuestro algoritmo 2 para resolver instancias de este tipo de problema nos damos cuenta de que el algoritmo nos puede decir que disparemos en una cabeza de una barrera específica porque terminó, pero resulta que dicha barrera puede no ser alcanzable con un disparo, como fue mencionado en un capítulo anterior. (Figura 3.14).

Peor aún, podemos tener dos configuraciones muy parecidas, cuyas gráficas de intervalos y conjuntos independientes máximos son iguales, y no podemos destruirlas con el mismo número de disparos usando nuestro algoritmo 2, ni con ningún otro (figura 4.1). Sin embargo, estableceremos qué tan grande es esta diferencia.

Por ejemplo, en la figura 4.1 supongamos que disparamos desde arriba y que tenemos una potencia de 3, es decir, $k = 3$. Con la configuración izquierda nos toma solo dos disparos, i.e. $d = 2$. Sin embargo, con la configuración derecha $d = 3$. (Recordemos que denotamos el número de disparos de una solución por d)

Figura 4.1: Mismas gráficas de intervalos y conjuntos independientes máximos, pero diferente número de disparos para destruirlas.

Al ver este problema nos damos cuenta de que nuestro algoritmo de potencia infinita es efectivo únicamente cuando la potencia k es lo suficientemente grande como para destruir la mayor cantidad de barreras. Es decir, si tenemos una cantidad de barreras adyacentes menor o igual a k podemos destruir todas de manera similar al caso de potencia infinita.
La forma en que pasaremos de una configuración a otra que sea como de potencia infinita es usando un algoritmo *ávido*.

Como se mencionó en la introducción, un algoritmo *ávido* (greedy) es aquél que toma decisiones localmente óptimas en cada paso, con la esperanza de encontrar la solución óptima global. No todos los algoritmos de este tipo encuentran el óptimo global pero en general llegan a una aproximación. Para nuestro caso, usaremos un algoritmo ávido tal que haga lo siguiente: Si encuentra k o más barreras adyacentes, dispara. Si encuentra menos de k, no dispara. Hay que hacer notar que un barrido no es necesario, de hecho podemos aplicar el algoritmo de manera aleatoria para ver dónde disparar.

La idea de usar este algoritmo es "desgastar" la configuración original, de modo que la llevemos a una donde se caiga en el caso de otra configuración tal que la potencia k sea tan grande que destruya cualquier conjunto de barreras adyacentes, como ocurre cuando tenemos potencia infinita.

El problema ahora es determinar qué tan lejos nos quedaremos del óptimo usando primero el algoritmo ávido para desgastar y después el de potencia infinita para terminar la tarea.

Decimos que OPT es el mínimo número de disparos para destruir un conjunto de barreras.

Tenemos la siguiente observación:

Observación 3. *La solución óptima OPT para un conjunto C es mayor o igual que la solución óptima OPT' para un subconjunto $C' \subseteq C$. Es decir, los disparos que destruyan a C no son más que los que destruyen a C'.*

Esta observación nos ayudará a demostrar que usando el algoritmo ávido y después el de potencia infinita podemos obtener una solución que es a lo más el doble de la solución óptima.

Teorema 7. *La aplicación del algoritmo ávido con potencia k y después el algoritmo 2 da una solución que tiene cuando mucho igual al doble de disparos que la solución óptima OPT.*

Demostración. Primero se aplica el algoritmo ávido sobre la configuración inicial C. El algoritmo consiste en disparar siempre que se puedan destruir k barreras, entonces si se hacen m disparos, al terminar este algoritmo habrá destruido

$k \times m$ barreras.

Como se disparó siempre que se podían destruir k barreras, tenemos que, por la aplicación del algoritmo 2, toda vertical cruza menos de k barreras. Entonces todas las barreras restantes se pueden destruir con disparos de potencia k.

Esta configuración $C' \subseteq C$, que es equivalente a la de potencia infinita, es la que nos queda después de aplicar el algoritmo ávido. Su solución la denotaremos por OPT'.

Entonces lo que queremos demostrar es que

$$OPT' + m \leq 2 \times OPT$$

Como vimos en el capítulo anterior, sabemos que OPT' será una solución óptima, por lo que ahora nos queda sólo demostrar que

$$m \leq OPT$$

Es decir, que el número de disparos que hace el algoritmo ávido es menor o igual que el óptimo. Por la observación 3, $OPT' \leq OPT$, entonces m, que es la solución para un subconjunto, no puede ser mayor a la solución para el conjunto entero (OPT).

□

La complejidad del algoritmo sigue siendo $O(n)$ si tenemos los extremos de las barreras ordenados. Sin embargo, veremos otro algoritmo que utilizando una complejidad mayor nos da una mejor aproximación bajo la condición de que podemos permutar verticalmente las barreras. Este algoritmo se origina del hecho de que algunas configuraciones no pueden destruirse óptimamente desde donde disparamos, que es arriba, pero sí desde abajo.

El algoritmo consiste básicamente en buscar un conjunto independiente máximo M con el algoritmo 2 de potencia infinita, poner las barreras que pertenecen a M hasta arriba, dispararle a M, y repetir. El algoritmo surge del hecho de que las barreras de M son las que guían los disparos, y que queremos aprovechar la potencia de cada disparo de modo de que cada disparo destruya a M junto con las barreras que se encuentren debajo. El problema resulta cuando al reacomodar nos siguen quedando barreras de menor tamaño abajo, ya que éstas son candidatas a pertenecer a un conjunto independiente máximo, y al dispararles a las que acomodamos hasta arriba, necesitaremos más disparos para destruir estas barreras cortas.

Algoritmo 4 Reconfiguración de una instancia y destrucción de sus barreras

Entrada: Los $2n$ extremos de las barreras.
Salida: Un conjunto M de verticales que cubre la configuración.

```
1: Sea M = ∅ el conjunto de verticales que cubrirá la configuración
2: Ordena los 2n puntos con respecto a su coordenada x
3: Sea p el punto más a la izquierda
4: mientras haya barreras sin destruir hacer
5:    si p es una cabeza entonces
6:       si si la barrera b que tiene a p como cabeza no está marcada entonces
7:          Agrega la barrera b al conjunto independiente máximo M
8:          Marca las barreras adyacentes a b
9:       fin si
10:   fin si
11:   si p es el último punto entonces
12:      Permuta todas las barreras de M arriba de todas las barreras
13:      Dispara en la cabeza de las barreras de M
14:      Quita la marca de las barreras restantes
15:      p el punto más a la izquierda
16:      Continúa el ciclo
17:   fin si
18:   p toma el valor del siguiente punto
19: fin mientras
20: devolver M
```

En particular este algoritmo funciona mejor con configuraciones donde las barreras más cortas están cubiertas por otras más largas, pero de manera muy simétrica. Por ejemplo, como en la figura 4.2.

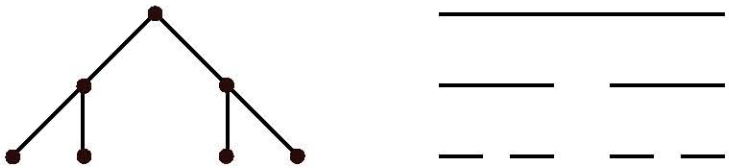

Figura 4.2: Configuración en forma de árbol.

El algoritmo funciona mejor con configuraciones de este tipo porque las hojas del árbol forman un conjunto independiente máximo, entonces si las pones arriba de la raíz, es seguro de que al disparar se destruirán las hojas junto con otras barreras.

Sin embargo, este algoritmo puede llegar a tener una complejidad de $O(n^2)$ con ciertas configuraciones. Por ejemplo, supongamos que tenemos potencia k y kn barreras colocadas en orden descendente verticalmente con respecto a su longitud (figura 4.3). En cada iteración, el algoritmo incluirá la barrera de más abajo en el conjunto independiente máximo M. Luego tomará las $kn - 1$ barreras, las recorrerá una posición hacia abajo y pondrá la barrera hasta arriba de todas y disparará, destruyendo esa barrera y las $k - 1$ que antes estaban en

el tope (las más grandes). En la siguiente iteración moverá otra vez las $kn-3$ barreras y pondrá la más chica hasta arriba. Entonces si tenemos kn barreras, haremos n iteraciones, dentro de las cuales haremos $kn, kn-2, kn-4, \ldots, k$ operaciones, lo cual quitando constantes nos da $O(n^2)$. Si hay muchas barreras y menor potencia, la complejidad empeora.

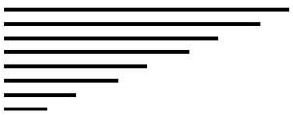

Figura 4.3: Configuración donde reacomodar constantemente tiene complejidad de tiempo cuadrática.

El problema con este algoritmo es que no funciona óptimamente con todas las configuraciones. Falla principalmente con aquellas configuraciones que tienen candidatos a un conjunto independiente máximo encerrados por otras barreras. Por ejemplo, en la figura 4.4 vemos cómo va quedando la configuración al aplicar el algoritmo 4. El algoritmo primero hace un barrido y detecta a d y c como elementos de un conjunto independiente máximo M. Después reacomoda a M en la parte superior y dispara en sus cabezas, destruyendo a M junto con a porque intersecta. Finalmente hace otro barrido y mete a b, la única barrera restante, a M. La reacomoda y la destruye. En total tomó tres disparos ($d = 3$). Sin embargo, la configuración pudo acomodarse mejor, como en la figura 4.5, donde $d = 2$.

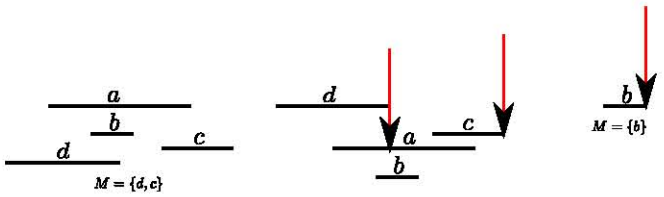

Figura 4.4: Ejemplo de ejecución donde no se destruye óptimamente la configuración.

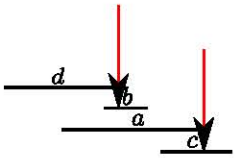

Figura 4.5: Ejemplo de reconfiguración donde se destruye óptimamente.

La razón principal por la que falla es que los elementos del conjunto independiente máximo M que encuentran no son siempre la mejor elección, ya que no significa que éstos cubran "completamente" a otras barreras. Por "completamente" nos referimos a que disparar en un elemento de M no asegura que destruirá

todas las barreras que intersectan con él, como ocurría con el caso de potencia infinita. Es justamente debido a que puede que la potencia no nos alcance para destruir todas. Entonces puede que otras barreras que marcamos para no ser incluidas en M nos pueden ayudar más, como es el caso de la configuración original en la figura 4.4, donde disparar en la cabeza de a destruye a c, y luego en la cabeza de d destruye a b.

Desde el capítulo anterior vimos que una de las claves para destruir óptimamente una configuración es aprovechar los conjuntos independientes máximos ya que los elementos de dicho conjunto guían el orden de los disparos. El problema es que candidatos a elementos de un conjunto independiente máximo quedan enterrados bajo más barreras, provocando que los disparos no lleguen hasta ellos y se tengan que hacer barridos y disparos adicionales. El siguiente algoritmo evita dicho problema.

La intención del siguiente algoritmo es hacer únicamente un barrido para hacer una sola reconfiguración del orden de las barreras. Al igual que el algoritmo anterior, solo mueve las barreras verticalmente. La idea es acomodar las barreras de tal forma que los candidatos a formar un conjunto independiente máximo queden arriba, para así, al disparar, destruirlos aprovechando siempre la potencia de cada disparo.
El algoritmo es el siguiente. Se toman los extremos de las barreras y se ordenan con respecto a su coordenada x, y luego con respecto a la y. Ordenar con respecto a x nos ayudará con el barrido, mientras que con respecto a y nos ayudará a acomodar las barreras. Se procede a hacer el barrido, y conforme se encuentra la cabeza de cada barrera se va a acomodar de la siguiente manera: la primera cabeza que se encuentre se pondrá en la posición más arriba de la configuración; la segunda cabeza se pondrá en la segunda posición más arriba, y así sucesivamente con todas las barreras. En la figura 4.6 vemos cómo estaba originalmente la configuración (arriba) y cómo queda reordenada (abajo). Hay que notar que como solo cambiamos el orden vertical, la gráfica de intervalos sigue siendo la misma. El algoritmo finalmente procede a disparar como el de potencia infinita: va haciendo un barrido de izquierda a derecha disparando en la cabeza de cada barrera que no está destruida.

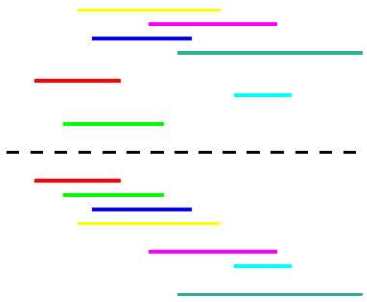

Figura 4.6: Ejemplo de reconfiguración en un solo barrido.

El algoritmo descrito antes es el siguiente (algoritmo 5).

Algoritmo 5 Reconfiguración y minimización de disparos con potencia constante.

Entrada: Los $2n$ extremos de las barreras y la potencia k
Salida: Un conjunto M de verticales que cubre la configuración.

1: Ordena los 2n puntos con respecto a su coordenada x
2: Ordena los 2n puntos con respecto a su coordenada y
3: Sea p el punto más a la izquierda
4: Sea b la barrera de más arriba
5: Sea $M = \emptyset$ el conjunto de verticales que cubrirá la configuración
6: **mientras** p sea un punto a la izquierda del último punto o sea el último punto **hacer**
7: **si** p es una cabeza **entonces**
8: Intercambia la posición y de la barrera que tiene como cabeza a p con la de la barrera b
9: b es la barrera abajo de p
10: **fin si**
11: p toma el valor del siguiente punto
12: **fin mientras**
13: Sea p el punto más a la izquierda
14: **mientras** haya barreras sin destruir **hacer**
15: **si** p es una cola **entonces**
16: p toma el valor del siguiente punto
17: **si no**
18: Agrega la vertical de p a M
19: Dispara en p destruyendo todas las barreras que se intersecten con la vertical que pasa por p
20: **si** hay barreras sin destruir **entonces**
21: p se vuelve punto más a la izquierda
22: **fin si**
23: **fin si**
24: **fin mientras**
25: **devolver** M

La configuración modificada por el algoritmo tiene diferentes propiedades.

Una vez que quedan reacomodadas las barreras (primer ciclo, pasos 6 a 12), la **invariante** del algoritmo es similiar a la del algoritmo 2 de potencia infinita (segundo ciclo, pasos 14 a 24): Sea f_i con $\{i = 1, 2, 3, \ldots, d\}$ una vertical del conjunto M. Al final de la iteración k, todo intervalo de la configuración inicial que tiene una cola $t(i)$ con $t(i) \leq f_k$, está cubierto por M. La invariante se sostiene por el lema 5, pero primero veremos otras propiedades para entenderlo mejor.

Podemos ver que las cabezas de las barreras no quedan cubiertas por ninguna otra desde arriba. En la figura 4.7 tenemos un ejemplo de una configuración original y luego la misma configuración modificada por el algoritmo 5. Los números representan la cantidad de barreras que bloquean la cabeza de las barreras. En la configuración original teníamos que romper barreras para llegar a algunas en específico. En cambio, con la modificación podemos llegar a cualquier barrera.

Siguiendo esto, le podemos ver una forma particular a la configuración mo-

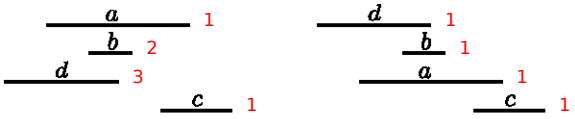

Figura 4.7: Al reconfigurar nos quedan descubiertas las cabezas de las barreras.

dificada. Se asemeja a una **escalera** (figura 4.8). Después de acomodar, el algoritmo hace un barrido de izquierda a derecha donde va disparando en cada cabeza que va encontrando. Esto lo podemos ver como si estuviera disparando en el fin de cada escalón, entonces como la potencia $k > 1$, destruirá al escalón y por lo menos un escalón debajo de él (a menos que sea el último, o que sea uno que no intersecta con otro, en estos casos no hay nada debajo). Sin embargo, si después de acomodar hiciéramos el barrido de derecha a izquierda disparando aún en las cabezas, cada disparo destruiría solamente una barrera, haciendo esta solución la peor después de acomodar. No es muy difícil ver que se forma una escalera pues estamos acomodando según el orden de aparición de las cabezas, y no por el tamaño de las barreras.

Este algoritmo nos muestra justamente cómo podemos acomodar una configuración de modo defensivo, haciendo que cada disparo desperdice potencia (suponiendo que sabemos que los disparos se harán de un extremo derecho o izquierdo al contrario).

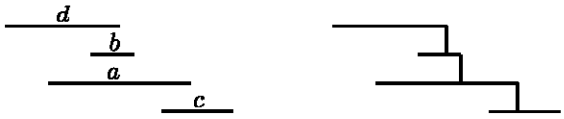

Figura 4.8: Al reconfigurar nos queda la forma de una escalera.

La complejidad del algoritmo sigue siendo $O(n \log n)$: Ordenar con respecto a x y luego con respecto a y nos toma $O(n \log n) + O(n \log n)$. El acomodar las barreras nos toma $O(n)$ pues revisamos cada una sólo una vez. Finalmente hacemos un último barrido en tiempo lineal: disparamos en cada cabeza que vemos y se destruye a la barrera junto con alguna más abajo. Dependiendo de la implementación, cada disparo puede llegar a tomar tiempo cuadrático si se elige una matriz de adyacencia como estructura de datos para guardar la gráfica, provocando que el algoritmo tenga una complejidad de $O(n^3)$ porque haríamos $O(n)$ disparos. Al final de este capítulo está la sección que contiene notas sobre la implementación. A continuación demostraremos algunos lemas para poder ver que el algoritmo es óptimo.

Lema 2. *Con potencia dos ($k = 2$), el algoritmo 5 no desperdicia potencia.*

Demostración. Como ya mencionamos, al acomodar la configuración queda de manera descendente en cuanto a la aparición de cabezas: la primera cabeza queda hasta arriba, la segunda en la segunda posición, y así sucesivamente; es decir, queda en forma de escalera. Ahora, supongamos que se tiene potencia dos, es decir, $k = 2$. Si todas las barreras son adyacentes, no se desperdicia potencia

excepto tal vez al disparar en la última barrera (esto depende de la paridad de barreras adyacentes). Esto es fácil de ver por lo siguiente. Supongamos que se tiene la configuración acomodada como escalera y se está disparando. De repente se tiene un disparo que rompió sólo una barrera pero todavía no se termina de destruir la configuración (figura 4.9, el conjunto de la izquierda en realidad ya estaría destruido pero en la figura lo incluimos simplemente para ejemplificar dónde se está disparando).

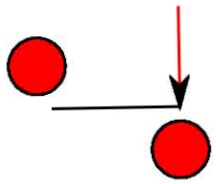

Figura 4.9: Al disparar sólo se destruye una barrera.

Si sólo se destruye una barrera pero no es la última, los posibles lugares de las barreras restantes serían arriba, abajo a la izquierda, y arriba a la derecha. Arriba no puede haber pues como ya se ordenó, esas posiciones están ocupadas por barreras cuyas cabezas se encontraron primero. Abajo a la izquierda antes de la cabeza de la que se está disparando tampoco, porque entonces la configuración no estaba ordenada por aparición de cabezas (figura 4.10).

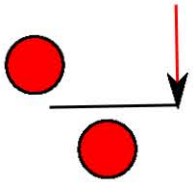

Figura 4.10: Al disparar sólo se destruye una barrera pero hay otras antes.

Entonces las barreras restantes sólo pueden estar abajo a la derecha, y si son adyacentes a la barrera en que se está disparando, el disparo alcanzará a una porque como ya fue mencionado, las cabezas de todas las barreras están descubiertas.

El único disparo que parece que desperdicia potencia es la última barrera, cuya cabeza está más a la derecha (si es que no ha sido destruida antes), pero este disparo sigue siendo necesario.

□

Lema 3. *Con potencia tres ($k = 3$), el algoritmo 5 no desperdicia potencia.*

Demostración. Todo lo anterior con $k = 2$ se mantiene, sin embargo puede pasar que se destruyen dos barreras y todavía no terminamos de destruir todas (figura 4.11) (una no, por lo mismo que con $k = 2$).

Si esto ocurre no hay problema tampoco porque era un disparo necesario: Supongamos que se tienen dos barreras,u y v (figura 4.11). Se fue destruyendo las barreras de modo que u no fue destruida con los disparos anteriores, entonces se debe disparar en ella porque es el siguiente disparo (es fácil ver que si no se

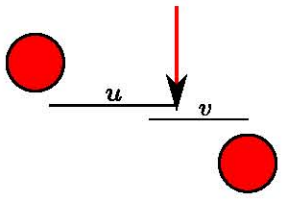

Figura 4.11: Al disparar con potencia tres sólo se destruyen dos barreras.

hace, no se destruyen todas las barreras). Podemos disparar no necesariamente en la cabeza y no se destruye a v, pero lo más conveniente es justamente destruirlas juntas (incluso es lo que hace el algoritmo cuando está disparando).

De hecho podemos ver las barreras que intersectan como un conjunto, mientras que las otras barreras que intersectan entre sí como otro. Entre ambos conjuntos hay una sola intersección, la cual es la de u y v, porque u pertenece al primero, y v al segundo (figura 4.12). El disparo que destruye a u y a v destruiría justamente la intersección entre ambos conjuntos, terminando de destruir al primer conjunto y comenzando a destruir al segundo.

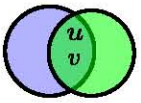

Figura 4.12: Lo más conveniente es destruir la intersección entre ambos conjuntos al mismo tiempo.

En el caso en que sólo quede una barrera y tengamos que disparar, es igual de necesario que cuando $k = 2$.

□

Lema 4. *Con potencia k, el algoritmo 5 no desperdicia potencia.*

Demostración. Si hacemos que $k = 4$, podemos ver que caen los mismos casos cuando $k = 2$ (última barrera, barrera no adyacente), y cuando $k = 3$ (un disparo no utiliza toda la potencia pero es necesario). Si seguimos aumentando la potencia, lo mismo se mantiene. Los lemas anteriores siguen. □

Lema 5. *En la fase de disparar, cuando el algoritmo 5 encuentra una cabeza, ya no quedan barreras arriba de ésta.*

Demostración. Supongamos que al llegar a una barrera l se tienen barreras arriba. Ya sabemos que el algoritmo no desperdicia potencias, así que para llegar a l tuvo que disparar destruyendo todo a su paso. Por construcción del algoritmo, no pueden quedarse barreras sin destruir arriba. □

Teorema 8. *El algoritmo 5 es óptimo.*

Demostración. Queda claro de que la ordenación de las barreras y el barrido nos da una complejidad de $O(n \log n)$. Ahora nos queda demostrar que los disparos son óptimos. Con los lemas anteriores tenemos que acomodamos la configuración de tal modo que cada disparo aprovecha toda la potencia, damos disparos

necesarios, y que se van destruyendo todas las barreras conforme se barre. Como inducción podemos verlo de la siguiente forma: Es claro que el primer disparo es óptimo pues no desperdiciamos potencia. Supongamos que el n-ésimo disparo es óptimo también (normalmente k-ésimo disparo pero para evitar confusión de notación usaremos n). Si el n-ésimo es óptimo, entonces el $n+1$ también lo es. Como demostramos que el algoritmo no va dejando barreras arriba conforme va disparando, entonces no se encontrarán cabezas antes de la cabeza de la barrera a la que se está a punto de disparar. Esto significa que cuando se esté por hacer el disparo $n+1$, no se tendrá nada antes, lo cual es lo mismo que como estaba la configuración cuando se hizo el primer disparo, por lo que es óptimo también como él. Por tanto, el n-ésimo disparo era óptimo también.

□

4.1. Notas de implementación

Una *matriz de adyacencia* A de una gráfica G con n vértices es una matriz (arreglo bidimensional) de tamaño $n \times n$ (cuadrada), cuyas celdas $a_{i,j}$ son el número de aristas del vértice i al vértice j. En la figura 4.13 podemos ver una gráfica no dirigida con su correspondiente matriz de adyacencia en la figura 4.14. La matriz es cuadrada porque tiene un renglón y una columna por vértice. Por ejemplo, en la figura 4.14, el primer renglón corresponde al vértice a. El 0 de la primera columna representa que no hay arista que salga desde a y llegue a él mismo. Los 1's de las columnas siguientes representan que existe una arista entre a y b, y otra entra a y c.

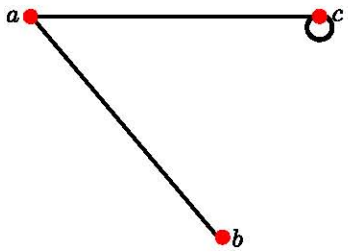

Figura 4.13: Gráfica no dirigida con un bucle

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Figura 4.14: Matriz de adyacencia correspondiente

Sin embargo, nuestras gráficas de intervalos son simples (no tienen aristas paralelas ni bucles, estos últimos son aristas que salen de un vértice y llegan a él mismo), por lo que la matriz tendrá únicamente 1s y 0s en sus celdas. Por ejemplo, si en la celda $a_{1,2}$ hay un 1, significa que existe una arista que une a los vértices 1 y 2. Si en vez de un 1 hubiera un 0, significa que no existe tal arista. Como nuestras gráficas no tienen dirección, la matriz resultante es *simétrica*. Es decir, $a_{i,j} = a_{j,i}$.
El problema de representar las gráficas en una computadora con esta estructura de datos es que el espacio es cuadrático ($O(n^2)$) porque es de tamaño $n \times n$. Entonces si hacemos n operaciones y en cada una de estas hacemos n^2 más operaciones, en total haremos $O(n^3)$. Esta estructura es buena si la gráfica es *densa*, es decir, que existen muchas aristas en la gráfica, ya que de no ser así,

tendremos mucho espacio desperdiciado con 0s.

La ventaja principal de esta estructura de datos es que si queremos saber si existe una arista que une a un par de vértices, hacemos una consulta en $O(1)$, porque solamente tenemos que ver el contenido de la celda $a_{i,j}$.

Sin embargo, si sabemos que nuestras gráficas serán *escasas* (no densas), una lista de adyacencia nos da una complejidad menor. Una *lista de adyacencia L* es una estructura de datos que consiste en una colección de listas no ordenadas, una para cada vértice de la gráfica. Cada lista describe el conjunto de vértices vecinos. Hay diferentes manera de representar esta estructura, por ejemplo en [1] se sugiere una implementación en la que los vértices son representados por índices numéricos. Su representación usa un arreglo indexado por el número de vértice, en el cual cada celda apunta a una lista simplemente ligada de los vértices vecinos del vértice (figura 4.16). Una lista simplemente ligada es un conjunto de nodos los cuales tienen dos campos: uno para información que se quiere almacenar (en nuestro caso el nombre o número de los vértices de la gráfica), y otro campo el cual apunta al nodo siguiente en la lista. Decimos que un nodo n_i "apunta" a otro nodo n_j si el campo de nodo de n_i tiene como referencia al nodo n_j. El último nodo de una lista simplemente ligada apunta al vacío.

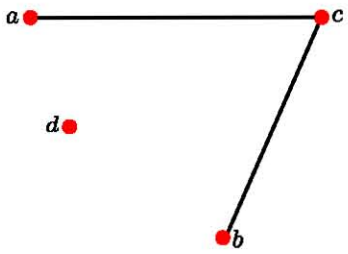

Figura 4.15: Gráfica no digirida y sin bucles

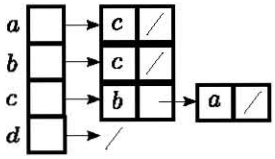

Figura 4.16: Lista de adyacencia correspondiente

La desventaja principal de esta estructura de datos es que las aristas no están representadas de manera explícita. Por ejemplo, si queremos saber si existe una arista que une a un par de vértices, tenemos que ir a la lista ligada de uno de los vértices y ahí recorrer la lista ligada buscando un nodo que contenga al otro vértice. La complejidad puede ser de $O(n)$ siendo n el número de vértices en una gráfica conectada (cada vértice está unido con todos los demás). Sin embargo, en una implementación puede ser menor si se van quitando los vértices de la lista. Otra desventaja, es que si la gráfica no es dirigida hay información duplicada. Por ejemplo, si el nodo n_i está unido por una arista con el nodo n_j, en la lista ligada de cada uno de ellos se encontrará al otro.

Volviendo a nuestro problema, con una lista de adyacencia se logra que cada disparo tenga una complejidad en tiempo menor a lineal, por lo que el barrido sigue siendo $O(n)$, logrando finalmente que el algoritmo 5 en total sigue tomando tiempo $O(n \log n)$.

Capítulo 5

Disparos de potencia variable

Antes de estudiar la siguiente variante de nuestro problema, veremos algunos conceptos necesarios para entenderlo mejor.

5.1. NP-completez

Los algoritmos que hemos visto hasta el momento en este trabajo han sido *algoritmos de tiempo polinomial*: con valores de entrada de tamaño n, la complejidad del peor caso es $O(n^k)$ para alguna constante k. Como en [1] se explica, es natural preguntarse si todos los problemas pueden ser resueltos en tiempo polinomial. La respuesta es no. Por ejemplo, existen problemas, como el famoso "Halting Problem" de Turing, que no puede ser resuelto por ninguna computadora sin importar cuando tiempo le demos. También hay problemas que pueden ser resueltos, pero no en $O(n^k)$ para alguna constante k. Sin embargo, existe una clase de problemas interesante. Estos problemas son llamados "NP-completos", cuyo estado es desconocido. No se ha descubierto un algoritmo de tiempo polinomial para un problema NP-completo, ni tampoco se ha probado que no puede existir un algoritmo de tiempo polinomial para cualquiera de ellos. Esta es la pregunta llamada P $\neq$ NP, y ha sido una de las más profundas y estudiadas desde 1971 ([15]). Algo curioso de los problemas NP-completos es que varios de ellos se parecen a problemas que sí tienen algoritmos de tiempo polinomial. Por ejemplo, el problema de encontrar el camino más corto en una gráfica dirigida $G = (V, E)$ se puede lograr en tiempo $O(VE)$. Sin embargo, encontrar el camino simple más largo entre dos vértices es difícil. De hecho determinar si una gráfica contiene un camino simple con al menos un número dado de aristas es NP-completo.

La clase P consiste en aquellos problemas que se pueden resolver en tiempo polinomial. Más específicamente, son problemas que pueden ser resueltos en $O(n^k)$ para alguna constante k, donde n es el tamaño o el número de bits de entrada. La clase NP consiste en aquellos problemas que son "verificables" en tiempo polinomial. Es decir, que si de alguna manera nos dan una solución, podemos verificar si es correcta en tiempo polinomial con respecto a los valores de entrada del problema. Veremos un ejemplo de esto al estudiar el problema de la suma del subconjunto.

Cualquier problema que está en P está también en NP ($P \subseteq NP$) ya que si un problema está en P entonces podemos resolverlo en tiempo polinomial sin siquiera necesitar que nos den una solución para verificar. La pregunta en sí es si $P \subset NP$.

Un problema está en la clase NP-completo (NPC) si está en NP y es tan difícil como cualquier problema en NP. Si cualquier problema NP-completo puede ser resuelto en tiempo polinomial, entonces cada problema en NP tiene un algoritmo de tiempo polinomial. En general se cree que $P \neq NP$. De ser este el caso, la relación entre P, NP, y NPC (NP-completo) es como la mostrada en la figura 5.1.

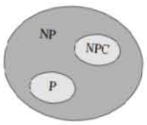

Figura 5.1: Relación entre P, NP, y NPC si $P \neq NP$

Para mostrar que un problema es NP-completo, normalmente se utilizan tres conceptos: problemas de decisión y de optimización, reducción, y otro problema NP-completo.

Muchos problemas de interés son *problemas de optimización*. En estos problemas tenemos solución factibles y queremos encontrar aquella solución que nos proporcione el mejor valor. Por ejemplo, el problema de la mochila que veremos más adelante (Knapsack problem) puede aproximarse. La NP-completez no se aplica directamente a problemas de optimización, sino a *problemas de decisión*. En éstos la respuesta es únicamente "sí" o "no".

A pesar de que mostrar que un problema es NP-completo es más de decisión, existe una relación entre los problemas de optimización y los de decisión. Normalmente podemos tomar un problema de optimización como uno de decisión al imponer una cota en el valor que queremos optimizar. Esta relación es conveniente cuando tratamos de mostrar que un problema de optimización es "difícil". Esto es porque de cierta manera el problema de decisión es "más fácil", o al menos "no más difícil". Es decir, si un problema de optimización es fácil, su problema de decisión relacionado es fácil también. Poniéndolo en términos de NP-completez, si podemos mostrar que un problema de decisión es difícil, estamos mostrando también que si problema de optimización relacionado también es difícil. Por tanto, a pesar de que ponemos atención en de decisión, la teoría de NP-completez a veces tiene implicaciones para los problemas de optimización.

Esta noción de mostrar que un problema no es más difícil o no más fácil que otro se mantiene incluso cuando ambos problemas son de decisión. Podemos aprovechar esto para probar NP-completez de la siguiente manera. Consideremos un problema A de decisión que queremos resolver en tiempo polinomial. Llamamos *instancia* a los valores de entrada de problema en particular. Ahora supongamos que tenemos un problema B de decisión diferente el cual ya sabe-

mos cómo resolver en tiempo polinomial. Finalmente, supongamos que tenemos un método o función que transforma cualquier instancia α de A en una instancia β de B con las siguientes características:

1. La transformación toma tiempo polinomial.
2. Las respuestas son las mismas: la respuesta para α es "sí" si y sólo si la respuesta para β es también "sí".

1. Dada una instancia α del problema A, usar un algoritmo de reducción de tiempo polinomial para transformarla en una instancia de β del problema B.
2. Correr el algoritmo de decisión de tiempo polinomial para B en la instancia β.
3. Usar la respuesta para β como la respuesta para α.

Este método es llamado *algoritmo de reducción* de tiempo polinomial (figura 5.2) y con él podemos resolver el problema A en tiempo polinomial:

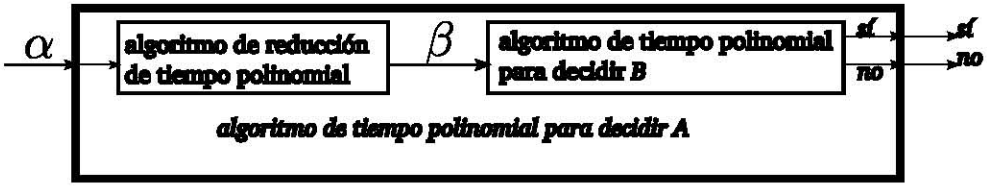

Figura 5.2: Algoritmo de reducción de tiempo polinomial.

Llamamos *función de reducción* a la función calculada por el algoritmo de reducción. Esta función f realiza un mapeo en tiempo polinomial tal que si un $x \in A$, entonces $f(x) \in B$ (figura 5.3).

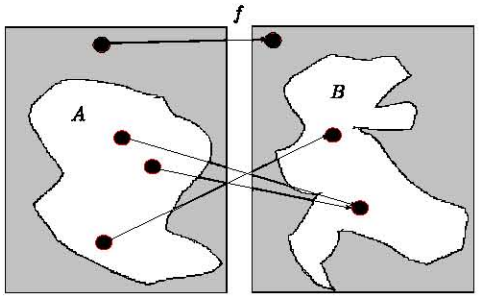

Figura 5.3: Función de reducción de tiempo polinomial.

Si estos pasos toman tiempo polinomial, los tres juntos toman ese tiempo también, por lo que tenemos una forma de decidir α en tiempo polinomial. En otras palabras, al reducir el problema A al problema B, usamos la "facilidad" de B para probar la "facilidad" de A.

La NP-completez se trata de mostrar qué tan difícil es un problema en vez de qué tan fácil. Entonces podemos usar reducciones de tiempo polinomial para mostrar que un problema es NP-completo. Supongamos que tenemos un problema de decisión A para el cual sabemos que no puede existir un algoritmo de tiempo polinomial. Supongamos también que tenemos una reducción de tiempo polinomial que transforma instancias de A a otro problema B. Con esto podemos usar una prueba por contradicción para mostrar que no puede existir un algoritmo de tiempo polinomial para B. Si suponemos lo contrario, es decir, que B sí tiene un algoritmo de tiempo polinomial, entonces tendríamos una manera de resolver el problema A en tiempo polinomial, pero esto contradice la suposición de que no hay algoritmo de tiempo polinomial para A.

Para la NP-completez no podemos asumir que no existe ningún algoritmo de tiempo polinomial para el problema A. El método de prueba es similiar en que podemos probar que B es NP-completo asumiendo que el problema A es también NP-completo.

El primer problema conocido como NP-completo es el *problema de satisfacibilidad booleana* (Boolean satisfiability problem). Este problema, abreviado como SAT, es aquél en el que se quiere determinar si existe una interpretación que satisfaga una fórmula booleana. Es decir, trata de establecer si las variables de una fórmula booleana pueden ser asignadas de tal suerte que la fórmula sea evaluada como verdadera. Si no se pueden asignar las formas de dicha manera, la función expresada por la fórmula es falsa para todas las posibles asignaciones de las variables. En este caso, se dice que no se puede satisfacer (unsatisfiable); en caso contrario se dice que se puede satisfacer (satisfiable). Por ejemplo, la fórmula $a \wedge \neg b$ se puede satisfacer porque se pueden asignar los valores $a = TRUE$ y $b = FALSE$, los cuales resultan en verdadero. En contraste, $a \wedge \neg a$ no se puede satisfacer. Este problema es muy importante ya que muchos problemas de decisión y optimización pueden ser transformados en instancias de SAT.

El *teorema Cook-Levin* (también conocido como el teorema de Cook) dice que el problema SAT es NP-completo. Es decir, cualquier problema en NP puede ser reducido en tiempo polinomial por una máquina de Turing determinística al problema de determinar si una fórmula booleana se puede satisfacer. Una consecuencia importante del teorema es que si existe un algoritmo determinístico de tiempo polinomial para resolver la satisfacibilidad booleana, entonces existe un algoritmo determinístico de tiempo polinomial para resolver todos los problemas en NP. Lo mismo se mantiene para cualquier problema NP-completo. Este es el problema "P versus NP" ([15]).

Lo que haremos en la siguiente sección del capítulo es justamente demostrar NP-completez de la siguiente variante del problema que estamos estudiando.

A continuación veremos una pequeña lista de problemas que son miembros de NP, y que de hecho son NP-completos ([14]).

Clan (clique). Dada una gráfica y un entero k, ¿existen en la gráfica k vértices que son adyacentes entre sí?

Número cromático (Chromatic number). Dada una gráfica y un entero k,

¿existe una forma de colorear los vértices con k colores de tal forma que los vértices adyacentes sean coloreados de manera diferente?

Cobertura de vértices (Vertex cover). Dada una gráfica y un entero k, ¿existe una colección de k vértices tal que cada arista está conectada a uno de los vértices de la colección?

Programación de examenes (Examination scheduling). Dada una lista de materias, una lista de conflictos entre ellas, y un entero k, ¿existe un horario de examenes que consista en k fechas tal que no existan conflictos entre las materias que tengan examenes en la misma fecha?

Recorrido cerrado (Closed tour). Dadas n ciudades y un entero k, ¿existe un recorrido de las ciudades de longitud menor a k que comience y termine en la misma ciudad?

Árbol generador Steiner recitlíneo (Rectilinear Steiner Spanning Tree). Dados n puntos en espacio euclidiano y un entero k, ¿existe una colección de líneas verticales horizontales de una longitud total menor a k de los cuales se generan los puntos?

Problema de la mochila (Knapsack). Dados n objetos cada uno con un peso y valor, y dos enteros k y m, ¿existe una colección de objetos con un peso total menor a k, y que tenga un valor total mayor a m?

5.2. Disparos de potencia variable

Continuando con la variante de nuestro problema, a continuación veremos una configuración todavía más realista que la estudiada en el capítulo anterior. En esta configuración seguimos teniendo resistencia mínima, pero ahora en vez de tener una cantidad ilimitada de disparos de potencia constante (la misma potencia para todos), tendremos potencia variable y un conjunto de disparos finito. Es decir, tendremos una colección de disparos con diferentes potencias y trataremos de aprovecharlos de la mejor manera. Hay que notar que podemos elegir cualquier disparo de la colección, es decir, que no estamos limitados a tomar siempre el primer disparo de la colección. Como ejemplo, podemos pensar en que tratamos de hacer una excavación y que tenemos diferentes tipos de taladros. Otro ejemplo podría ser que queremos romper tabiques apilados con nuestra mano (como en karate) y podemos ponerle diferente fuerza a cada golpe.

En este caso queremos responder si se puede destruir una configuración que nos es dada utilizando la colección de disparos que nos es ofrecida. Este no es un problema de optimización, sino de decidibilidad.

Podemos utilizar un algoritmo ávido que dispara donde parece que es el mejor lugar para disparar, pero aún así falla. El algoritmo ávido barre horizontalmente para contar cuántas barreras hay arriba de cada cabeza, incluyéndo a éstas últimas también en el conteo. Al terminar un barrido, cada cabeza ten-

drá un contador asociado. Nos referiremos al contador de una barrera como su *profundidad*. Después de un barrido, el algoritmo dispara en la cabeza cuya profunidad pueda ser emparejada con un disparo de la misma potencia. Si este emparejamiento exacto no es posible, se toma el disparo con la potencia más cercana por arriba a la profundidad. Es decir, si tenemos una barrera con profunidad tres y el conjunto de disparos $\{2, 4\}$, el algoritmo elige el disparo de potencia cuatro porque destruirá la barrera. Después de hacer un disparo, el algoritmo vuelve a barrer. Hace las iteraciones necesarias hasta que no queden barreras (o disparos, como veremos a continuación).

La idea del algoritmo es emparejar cada profunidad con un disparo de potencia igual. Se hace sólo un disparo después de cada barrido porque cada disparo cambia la configuración, y las profunidades no reflejan la configuración actual. El algoritmo no dispara mientras barre porque la idea es tener una "imagen global" de la configuración, y así poder elegir el disparo que parece el mejor. De hecho esto permite que el primer disparo no sea necesariamente en la cabeza que está más a la izquierda. Desafortunadamente, el algoritmo tiene una complejidad de $O(n^2)$ (si no hay intersecciones) y no siempre funciona.

A continuación veremos un ejemplo. Supongamos que tenemos el conjunto de disparos $\{2, 2\}$ para el siguiente par de figuras.

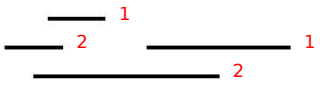

Figura 5.4: Una configuración donde el conteo ávido funciona.

Figura 5.5: Una configuración donde el conteo ávido no funciona.

Para la figura 5.4 funciona, pero para la figura 5.5 falla a pesar de que un disparo se va a aprovechar exactamente para la barrera con profunidad dos. Como podemos ver en la figura 5.5, tampoco nos sirve tomar la heurística de disparar con la mayor potencia a la cabeza con el mayor contador.

La reducción de un problema conocido al nuestro nos ayuda a saber más del nuestro. Una *reducción* es un algoritmo que transforma un problema en otro problema. Normalmente se aplica para mostrar que el segundo problema es por lo menos tan difícil como el primero. La reducción es en un sentido, por lo que los algoritmos que resuelven el primer problema no necesariamente sirven para resolver el segundo. Sin embargo, los algoritmos que resuelven el segundo sí nos podrían ayudar a resolver el primero. En nuestro caso, el primer problema sería el ya conocido (que veremos a continuación), y el segundo sería el de potencia variable que estamos estudiando. En la figura 5.6 el círculo de la izquierda representa el problema conocido, y el círculo de la derecha el de potencia variable. Para ver más a detalle sobre el tema de reducciones, ver la sección anterior. A continuación haremos la reducción a nuestro problema.

Figura 5.6: Reducción de un problema a otro.

Primero veremos diferentes problemas hasta que lleguemos a aquel cuya reducción hacia nuestro problema es directa.

El primer problema es el *problema de la mochila* (Knapsack problem [13]). Pertenece a la optimización combinatoria. En este problema tenemos una mochila la cual soporta un peso definido. Tenemos también un conjunto de objetos que tienen peso y valor cada uno. La idea es llenar la mochila de tal forma que maximizemos el valor total de los objetos en la mochila pero sin exceder el peso que resiste. Existe la versión de decisión, en la cual queremos que un algoritmo nos responda "sí" o "no", mientras que en la versión de optimización se trata de llenar la mochila lo más que se pueda. Se sabe que este problema y sus variantes son NP-completos, es decir, que podemos revisar rápidamente si una solución dada es correcta, pero el proceso de calcular una es difícil. Existen diferentes versiones de este problema. La versión de 0-1 restringe a que no podemos tener copias de los objetos. Es decir, cada objeto es único. *El problema de la mochila 0-1* se puede formular de la siguiente manera: Sean n objetos, x_1 a x_n donde x_i tiene un valor v_i y peso w_i, ambos positivos. El peso máximo que puede resistir la mochila es de W. Su formulación es la siguiente: Maximizar $\sum_{i=1}^{n} v_i x_i$ sujeta a $\sum_{i=1}^{n} w_i x_i \leq W$, $x_i \in \{0, 1\}$. Es decir, lo que se quiere es maximizar la suma de los valores de los objetos dentro de la mochila, de tal manera que la suma de los pesos sea menor o igual al peso que resiste la mochila.

El problema de la mochila acotado quita la restricción de que los objetos son únicos. Ahora restringe el número x_i de copias a un valor entero c_i. Puede ser formulado como sigue: Maximizar $\sum_{i=1}^{n} v_i x_i$ sujeta a $\sum_{i=1}^{n} w_i x_i \leq W$, $x_i \in \{0, 1, \ldots, c_i\}$.

El problema de la mochila no acotado quita la restricción en el número de copias de cada objeto, es decir, podemos tomar la cantidad que sea de copias de cada elemento, siempre y cuando la mochila aguante.

Con programación dinámica podemos resolver las variaciones del problema no acotado y 0-1. Además de la programación dinámica, existen algoritmos de aproximación. Por ejemplo, existe un algoritmo ávido el cual obtiene un factor v_i/w_i para cada objeto. Este factor representa qué tanto conviene meter el objeto a la mochila. El algoritmo luego ordena estos factores y va metiendo los objetos comenzando por los que más convengan hasta completar la carga de la mochila. Este algoritmo ávido lo aplicamos muchas veces en diferentes video juegos en los que jugamos con un personaje que tiene un inventario y cada objeto que recoge tiene un precio de venta y peso (e.g. The Elder Scrolls V: Skyrim, The Witcher 2: Assassins Of Kings). Si ponemos más peso del que puede soportar el inventario, el personaje se mueve lento y no puede realizar algunas acciones. Por esto los jugadores tiran los objetos que menos nos convenga conservar, mientras que conservan los más valiosos que puedan cargar.

La reducción de este problema al nuestro no se ve muy clara todavía. Sin embargo, a continuación veremos un caso especial del problema de la mochila que se parece más al que estamos tratando de destrucción de barreras.

El problema de la suma del subconjunto (Subset sum problem) es un caso especial de decisión de la versión 0-1 del problema de la mochila. Aquí cada valor es igual a su peso, es decir, $v_i = w_i$. En este problema tomamos al conjunto de objetos como un conjunto de números enteros positivos S. Lo que queremos saber es si existe un subconjunto $S_i \subseteq S$ tal que la suma de sus elementos dé exactamente como resultado la capacidad W de la mochila. Éste último número lo denotaremos ahora por x. Este problema sigue siendo NP-completo.

Existe un algoritmo que nos puede dar la respuesta exacta, sin embargo es muy lento. El algoritmo consiste en explorar todos los posibles subconjuntos que podemos formar con S. Hacer esto tiene una complejidad de $O(2^n)$, es decir, exponencial. Sin embargo, en [1] se presenta un algoritmo de aproximación en tiempo polinomial.

Existe una reducción del problema de la suma de subconjunto al nuestro. Consiste en tomar cualquier instancia del problema de la suma de subconjunto a partir de ella construir una instancia de nuestro problema. Lo que hacemos es tomar el conjunto S y verlo como si fuera el conjunto de disparos. Luego tomamos el número x (anteriormente era la capacidad W de la mochila) y ponemos x barreras que se intersecten todas entre sí. Si en nuestro problema puediéramos encontrar aquél subconjunto $S_i \subseteq S$ con el cual destruimos exactamente la configuración de x barreras, entonces estaríamos resolviendo también el problema de la suma de subconjunto.

La reducción anterior es interesante. Sin embargo veamos una reducción todavía más directa de otro problema, conocido como el *problema de partición* (Partition problem). Es un caso especial del problema de la suma de subconjunto. La x del problema anterior se interpreta como la mitad de la suma de todos los elementos de S. El problema es aquél que trata de decidir si el conjunto se puede partir en dos subconjuntos S_1 y S_2 de S, de tal forma que la suma de los elementos de S_1 sea igual a la suma para S_2.

En la versión de optimización de este problema se trata de minimizar la diferencia de las sumas. Este problema es NP-completo. De hecho lo podemos ver como la manera en que se organiza un grupo de amigos para jugar futbol ([10]). Primero se eligen dos capitanes de más o menos la misma habilidad. Después éstos se turnan para elegir a los miembros de su equipo. Esta elección normalmente se hace tomando en cuenta la habilidad de cada uno de los jugadores. Si se quiere un juego balanceado, cuando el primer equipo elige un jugador, el otro equipo elige otro jugador con una habilidad muy similar al jugador que acaba de ser elegido. Cuando todos los jugadores pertenezcan a algún equipo se termina el proceso de selección. Si sumamos las habilidades de los jugadores de un equipo veremos que esta suma es muy similar a la del otro equipo. Hemos descrito justamente un algoritmo ávido para resolver el problema de minimización.

La reducción del problema de la partición consiste en tomar una instancia de él y con ella construir "rápidamente" una instancia de nuestro problema. En

este caso es directo por lo siguiente: Tenemos una instancia del problema de la partición donde hay un conjunto S de enteros positivos. Lo que queremos saber es si podemos encontar a los subconjuntos S_1 y S_2. Entonces para hacer la reducción tomamos a S como el conjunto de disparos, y hacemos la suma de sus elementos igual a $2m$. A esta suma la dividimos entre dos y nos queda m. Sin pérdida de generalidad supongamos que la división fue exacta. Entonces podemos construir una configuración tal que apilamos un conjunto C_1 de m barreras que tienen una intersección común no vacía, y otro conjunto C_2 también de m barreras que se intersectan de la misma manera. Además, no hay intervalo en C_1 que tenga intersección no vacía con algún intervalo de C_2 (figura 5.7). De este modo ya tenemos una instancia y una reducción. Es fácil ver que resolver nuestro problema resolvería también el del problema de la partición, por lo que podemos decir que nuestro problema es al menos tan difícil como el de la partición.

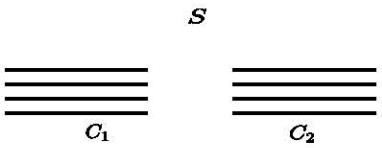

Figura 5.7: Reducción del problema de la partición al de disparos de potencia variable.

Por ejemplo, supongamos que $S = \{3, 6, 9, 11, 4, 7\}$, $2m = 40$, y el número de barreras adyacentes en C_1 y C_2 es $m = 20$. Es fácil verificar que los subconjuntos $S_1 = \{9, 11\}$, y $S_2 = \{3, 6, 4, 7\}$ son solución (única en este caso). Pero encontrar esta partición en general es difícil.

Revisar todos los posibles subconjuntos resulta en un algoritmo con complejidad exponencial $O(2^n)$. Es una solución exacta pero extremadamente lenta.

El problema sigue igual de díficil incluso cuando tenemos la libertad de reacomodar las barreras como en el capítulo anterior. Después de reacomodar y obtener la forma de "escalera", puede que no todas las barreras intersecten con todas por lo que sigue siendo difícil saber qué disparo es el mejor.

Capítulo 6

Conclusiones

Con lo presentado en este trabajo es fácil idear un plan para destruir una configuración de barreras. Necesitamos el conocimiento de la configuración y el tipo de disparos que tenemos. Por ejemplo, aunque no tengamos disparos de potencia infinita pero sí de una potencia lo suficientemente grande tal que puede atravesar cualquier punto debajo de la configuración, entonces es como si tuviéramos de potencia infinita. Por esto podemos aplicar fácilmente nuestro algoritmo, y de manera óptima destruimos toda la configuración en $O(n \log n)$.

Si nos otorgan el poder de acomodar la configuración como queramos, podemos hacerlo de manera que los disparos los hagamos de manera óptima al aprovechar la potencia de cada uno y hacer únicamente disparos necesarios.

Si tenemos un conjunto finito de potencias diferentes, podemos destruir exactamente la configuración en el enorme tiempo exponencial de $O(2^n)$ pues tendríamos que probar todos los subconjuntos posibles con el conjunto de potencias. Esto es debido a que con las reducciones sabemos que estos problemas son por lo menos tan difíciles como el problema de la partición, el cual es NP-completo.

La tabla 6.1 muestra un resumen de las complejidades de los algoritmos presentados para cada variante del problema.

6.1. Trabajo futuro

Mientras trabajábamos en el problema de disparos de potencias diferentes, surgió la variante de barreras con resistencias diferentes. Es decir, supongamos que definimos que una barrera puede ser destruida únicamente si cierta cantidad de disparos pasan a través de ella. Por ejemplo, supongamos que una barrera tiene resistancia cinco. Entonces cinco disparos de potencia uno la pueden des-

Variante	Configuración Original	Reconfiguración
Disparos de potencia infinita	$O(n \log n)$	$O(n \log n)$
Disparos de potencia constante	$O(n \log n)$	$O(n \log n)$
Disparos de potencia variable	NP-completo	NP-completo

Cuadro 6.1: Resumen de complejidades de algoritmos para cada variante del problema

truir; o un disparo de potencia tres y uno de dos; o un disparo de potencia cuatro y uno de una, etc. Entonces podríamos tomar las variantes estudiadas en este trabajo y añadir esta condición (la variante con potencia infinita sería trivial). Intuitivamente podemos decir que serían problemas NP-completos, por lo que sería bueno explorar la posibilidad de desarrollar algoritmos de aproximación.

Bibliografía

[1] Thomas H. Cormen. Charles E. Leiserson. Ronalds L. Rivest. Clifford Stein, *Introduction to Algorithms.*. The MIT Press, 2009.

[2] Gupta, U. I. and Lee, D. T. and Leung, J. Y.-T., *Efficient algorithms for interval graphs and circular-arc graphs.*, Networks, Vol. 12, No. 4, pp. 459-467, 1982.

[3] Nandy, Subhas C., Krishnendu Mukhopadhyaya, and Bhargab B. Bhattacharya. *Shooter Location Problems.*, 1996.

[4] Cao An Wang and Binhai Zhu, *Shooter Location Problems Revisited.*, Proc. 9th Canad. Conf. on Computational Geometry, pp. 223-228, 1997.

[5] Chaudhuri, Jeet, and Subhas C. Nandy, *Generalized shooter location problem.*, Computing and Combinatorics. Springer Berlin Heidelberg, pp. 389-399, 1999.

[6] Stefan Langerman, *On the Shooter Location Problem: Mantaining Dynamic Circular-Arc Graphs.*, 2000.

[7] H. Edelsbrunner, H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzl, D. Wood, *Stabbing Line Segments.*, 1982.

[8] H. Edelsbrunner, M. H. Overmars, D. Wood, *Graphics in flatland: A case study.*, 1981.

[9] Matthew J. Katz, Joseph S. B. Mitchell, Yuval Nir, *Orthogonal Segment Stabbing.*, 2004.

[10] Brian Hayes, *The Easiest Hard Problem*, American Scientist, Vol. 90, No. 3, pp. 113, 2002.

[11] Richard A. Lovett, *Scientists to Drill Earth's Mantle, Retrieve First Sample?*, National Geographic News, 2011.

[12] Tome Levitt, *The $1 billion mission to reach the Earth's mantle*, CNN, 2012.

[13] David Pisinger, *Algorithms for Knapsack Problems.*, Ph.D. thesis, February 1995.

[14] Forbes D. Lewis, *The Classes P and NP.*, 1996.

[15] Stephen Cook, *The complexity of theorem proving procedures.*, Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151-158, 1971.