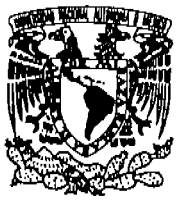


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

"Construyendo k-generadores y Árboles de Steiner en Redes Inalámbricas Móbiles Ad-hoc"

TESIS

que para obtener el título de:

Licenciado en Ciencias de la Computación

presenta:

Francisco Javier Escalona González

Director de tesis: Dr. Jorge Urrutia Galicia

2008

FACULTAD DE CIENCIAS

División de Estudios Profesionales

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales
Facultad de Ciencias
P r e s e n t e .

Por este medio hacemos de su conocimiento que hemos revisado el trabajo escrito titulado:

"Construyendo k-generadores y Árboles de Steiner en Redes Inalámbricas Móbiles Ad-hoc"

realizado por **Escalona González Francisco Javier,** con número de cuenta **097437938,** quien opta por titularse en la opción de **Tesis** de la licenciatura en **Ciencias de la Computación.** Dicho trabajo cuenta con nuestro voto aprobatorio.

Tutor(a) Propietario	Dr.	Jorge Urrutia Galicia	
Propietario	Dr.	Sergio Rajsbaum Gorodezky	
Propietario	Dr.	Criel Merino López	
Suplente	M. en C.	Mario Lomelí Haro	
Suplente	L. en C.C.	Ruy Fabila Monroy	

A t e n t a m e n t e
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Ciudad Universitaria, D.F., a 8 de junio del 2007.
EL COORDINADOR DEL COMITÉ DE TITULACIÓN
DE LA LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

FACULTAD DE CIENCIAS
DIRECCIÓN ADMINISTRATIVA

DR. JOSÉ DE JESÚS GALAVIZ CASAS

Señor sinodal: antes de firmar este documento, solicite al estudiante que le muestre la versión digital de su trabajo y verifique que la misma incluya todas las observaciones y correcciones que usted hizo sobre el mismo.

Agradecimientos

Este trabajo está dedicado a mis maestros. Algunos de ellos me han enseñado más de lo que se proponían y otros nunca se dieron cuenta que me estaban enseñando. De todos modos, a todos ellos les tengo un profundo agradecimiento por lo que me han enseñado.

En primer lugar, a Lourdes, Héctor y Héctor, cuya paciencia y sabiduría permea todo lo que soy. Me siento muy afortunado de conocerlos y que hayan compartido su mundo y sus familias conmigo. Por siempre gracias.

A Jazmín, que cambió mi vida de mil formas distintas y con quien he podido compartir todo. De pocas personas he aprendido tanto.

Los amigos más antiguos e incondicionales que tengo. Crecimos y nos hicimos juntos. A *la banda*. Mis hermanos y hermana por elección.

A Karla, por darme una de las oportunidades más importantes de mi vida al permitirme dar clase con ella. Por toda la experiencia transmitida, gracias.

A mis colegas, que nunca han dejado de sorprenderme y mostrarme la diversidad de opiniones y formas que puede tener un computólogo. Alfredo, Chandra, Ruy, Griselda, Grecia, John, Josafat, Mary, Gustavo, Victor, Rafael, Marco, todos los demás, y todos los que dejamos en el camino. A cada uno debo mi agradecimiento en mayor o menor medida.

A la UNAM. Mi cuerpo ya lleva basalto del pedregal y en ocasiones mis ojos perciben el mundo como en murales de Siqueiros, Rivera y O'Gorman. Esta institución fantástica que me ha cobijado y moldeado tiene mi eterno agradecimiento. Siempre irá conmigo.

Finalmente, aunque en un lugar muy especial, a mi más grande maestro en la academia, el Dr. Jorge Urrutia, quien siempre ha sido en extremo generoso con su conocimiento y con su tiempo, e incluso cuando fue necesario con sus recursos. Sus enseñanzas me han permitido conocerme mejor. Por su ejemplo soy un mejor alumno y maestro.

Índice general

Índice de figuras

Introducción

Los problemas de ruteo se encuentran entre los más estudiados en la actualidad. Sus aplicaciones son variadas, pero la principal razón para su apogeo se encuentra en el fuerte impulso que se le ha dado a la tecnología que hay detrás de las comunicaciones modernas. Contamos con redes de comunicación de muchos tipos, que se conectan con dispositivos muy variados: teléfonos fijos, celulares, televisiones, computadoras personales, satélites y demás, conectados a través de cables de cobre, ondas de radio, fibra óptica, etc. Cada tecnología con su propia idiosincracia.

Esta riqueza de elementos y técnicas se ve reflejada en la variedad de problemas propuestos y la genialidad de algunas de sus soluciones. Aun así, el problema subyacente, básico, que define los problemas de ruteo, consiste en encontrar la *mejor* manera en que se puede llevar un paquete de un lugar a otro, economizando los recursos disponibles.

Por ejemplo, si estamos en algún lugar de una ciudad y queremos ir manejando a otro, buscaremos de entre las rutas posibles aquella que nos tome menos tiempo. Este es un problema de ruteo.

Algo un poco más complicado es cuando se tienen múltiples fuentes o múltiples destinos. Extendiendo el ejemplo anterior, si tenemos que pasar a recoger a varios amigos en distintos puntos de la ciudadd no solo tenemos que encontrar la ruta que nos ahorre más tiempo, sino que mientras nosotros viajamos nuestros amigos pueden caminar un poco para encontrarse en algún punto intermedio. Esto reduciría el número de paradas que hay que realizar y posiblemente el tiempo total del recorrido. Como se puede observar, calcular una solución óptima se vuelve una tarea más difícil. En este trabajo nos enfocaremos principalmente en este segundo tipo de problema de ruteo.

Utilizaremos herramientas de la Teoría de Gráficas, Geometría Computacional y Teoría de la Complejidad para nuestro estudio. Se recomienda que el lector tenga un conocimiento al menos general de estas tres disciplinas, para lo cual lo referiremos a textos introductorios en cada área. Como mínimo, es esencial un conocimiento básico de Análisis de Algoritmos y las definiciones principales de Teoría de Gráficas.

Distribución del resto de la obra

Lo primero que haremos es establecer claramente el problema que nos interesa estudiar: la construcción de Árboles de Steiner. Nuestra principal herramienta para esto es la teoría de gráficas, que nos permitirá establecer algunas propiedades básicas de este objeto. Hecho esto propondremos un algoritmo que busque la solución al problema. El capítulo 1 trata estos temas y concluye con un primer análisis del algoritmo, donde mostraremos que falla en cumplir su objetivo.

En el capítulo 2 haremos un estudio sobre la dificultad intrínseca al tratar de resolver el problema del Árbol de Steiner. En particular probaremos que el problema es NP-completo. Como consecuencia de este resultado nuestra mejor opción será buscar una manera de aproximar la solución.

Con esta información, en el capítulo 3 regresaremos a estudiar el algoritmo propuesto inicialmente, pero ahora con el enfoque de descubrir si la gráfica que construye es una aproximación. Para esto lo replantearemos a través de su relación con el problema del Árbol Generador de Peso Mínimo. También estudiaremos algunas otras propiedades más avanzadas de los árboles de Steiner, mediante las cuales plantearemos algunas heurísticas útiles.

Como fruto de este estudio obtendremos un mecanimo para construir una 2-aproximación al árbol, así como algunas herramientas para generar una solución exacta en algunos casos especiales.

Luego, en el capítulo 4 realizaremos el análisis de complejidad de los algoritmos y técnicas que desarrollamos antes. Aquí haremos amplio uso de varios resultados de Geometría Computacional.

Una vez establecidos estos resultados introduciremos la complejidad de trabajar sobre redes móviles. En el capítulo 5 haremos una introducción a los algoritmos que tienen que trabajar sobre este tipo de redes. Mostraremos algunos ejemplos importantes y describiremos cómo se pueden adaptar nuestros resultados para ser usados en este ambiente, donde el ruteo se hace de forma distribuída y local. Nuestra principal herramienta será la construcción de un k-generador, que es una subgráfica que aproxima las distancias entre vértices en la gráfica original.

Finalmente en el capítulo 6 hablaremos, a manera de conclusión, sobre los resultados del trabajo y algunos problemas abiertos que se pueden seguir atacando.

Capítulo 1

Conectar Nodos Eficientemente

En este capítulo definimos el problema que vamos a tratar durante el resto de la obra. En la primera sección presentamos un ejemplo ilustrativo, así como una definición matemática formal.

En la segunda sección se muestran algunas observaciones preliminares. El objetivo es construir cierta intuición que sirva para entender mejor los resultados más avanzados.

Luego, en la tercera sección, utilizamos la experiencia de la Geometría Computacional con otros problemas similares para plantear un algoritmo que busca la solución. El algoritmo que obtendremos resultará encontrar solamente *aproximaciones*, lo cual mostraremos en capítulos posteriores. Por lo pronto en la sección cuatro daremos un ejemplo que ilustra como el algoritmo falla. Aun así, estas aproximaciones nos serán de utilidad.

1.1. Definición del Problema

Supongamos que un grupo de personas necesita comunicarse mutuamente en un momento determinado, por ejemplo, para realizar una videoconferencia, y que no se encuentran todos en el mismo lugar. Si tienen las herramientas electrónicas apropiadas, pueden conectarse todos a una red y llevar a cabo su comunicación por ese medio. Debemos tratar de usar la menor cantidad de recursos que sea posible, pero permitiendo que se puedan intercambiar mensajes entre ellos. La idea es no cargar innecesariamente de trabajo a otras computadoras, por eso mientras menos recursos de la red usemos, mejor.

La solución a este problema es nuestro objeto de estudio.

Formalizando Conceptos

En el apéndice A damos todas las definiciones necesarias para los conceptos de teoría de gráficas que usaremos a continuación. Consiste en una mera introducción al tema, por lo que para un tratamiento más profundo se recomienda revisar [7, 5].

La mejor manera de modelar una red es usando gráficas. Cada vértice representa un objeto en la red y trazamos una arista entre dos vértices si existe una comunicación directa entre ellos.

En otras palabras, el problema se puede plantear de la siguiente manera: dada una gráfica conexa $G = (V, E)$ y un subconjunto $S \subset V$ de *vértices terminales*, encontrar una subgráfica conexa de G con el menor número de vértices posible, que contenga a todos los elementos de S.

Cuando la subgráfica minimiza también el número de aristas se le conoce en la literatura como *árbol de Steiner*, por razones que veremos más adelante.

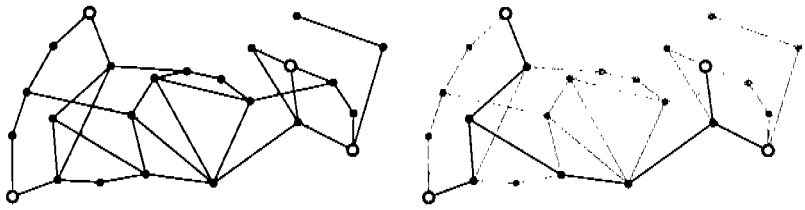


Figura 1.1: Una gráfica con vértices terminales marcados, y una posible solución.

En la figura 1.1 se muestra una gráfica con los vértices terminales pintados con blanco y la solución al problema para este caso. De aquí en adelante haremos esta convención con los colores en las figuras, pintando de negro a los vértices no-terminales.

1.2. Propiedades de la Solución

Para comenzar nuestro estudio, son necesarias algunas definiciones básicas. Las *hojas* de un árbol son los vértices de grado 1 y el resto son *vértices interiores*. De estos últimos, a aquellos que tengan grado superior a 2 les llamaremos *vértices de ramificación*. La figura 1.2 ejemplifica todos estos conceptos.

En lo que sigue, $G = (V, E)$ será una gráfica, S un subconjunto de V, y u, v dos elementos cualesquiera de V.

Definición 1.1. La *distancia* entre u y v está dada por el mínimo de las longitudes de los caminos entre estos vértices, y la denotamos por $\delta_G(u, v)$. Haciendo un abuso de notación, indicamos de la misma manera la distancia entre un vértice y un conjunto de vértices, es decir

$$\delta_G(u, S) = min\{\delta_G(u, s) | s \in S\}$$

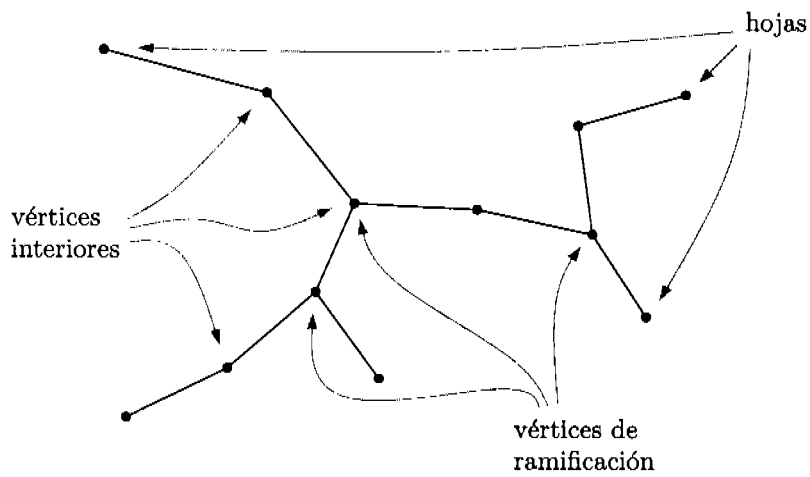


Figura 1.2: Un árbol y sus distintas partes.

Si no existe ninguna trayectoria entre u y v, la distancia entre ellos no está definida. Esto lo denotamos como $\delta_G(u, v) = \infty$.

Cuando sea claro por el contexto la gráfica sobre la que se está trabajando, omitiremos el subíndice. Esta distancia indica el mínimo número de saltos de aristas que hay que hacer para ir de un vértice a otro.

Definición 1.2. Sea $k \in \mathbb{N}$. La *k-vecindad* de v, denotada $N_k(v)$, son todos aquellos vértices de la gráfica que se encuentran a distancia a lo más k de v. En símbolos,

$$N_k(v) = \{x \in V | \delta(v, x) \leq k\}$$

1.2.1. Estructura de un Árbol de Steiner

Hay que observar que al resolver el problema es posible que algunas soluciones contengan ciclos, pues no se está minimizando la cantidad de aristas. En la figura 1.3 se muestra un ejemplo de esto. La gráfica completa cumple las condiciones de la solución para el conjunto de vértices terminales y sin embargo contiene un ciclo.

Si se tiene una gráfica conexa con ciclos, es posible encontrarle un árbol generador mediante un procedimiento sencillo: si existe algún ciclo, le quitamos una de sus aristas para romperlo y repetimos este proceso hasta que se eliminen todos los ciclos. Este método mantiene la conexidad de la gráfica original.

Por simplicidad, y dado que el método anterior es sencillo y siempre se puede aplicar, de aquí en adelante daremos por hecho que la solución al problema es un árbol, es decir que minimiza también la cantidad de aristas. Pueden existir varios árboles de Steiner para S en G, pero todos tienen el mismo número de vértices.

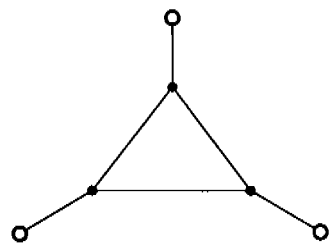


Figura 1.3: La gráfica buscada puede tener ciclos.

Observación 1.1. Todas las hojas en un árbol de Steiner son elementos de S.

Esto se sigue del hecho de que si un vértice no-terminal fuera hoja del árbol, lo podríamos podar. El resultado sería otro árbol que, además de contener a S, usaría menos vértices de G, lo cual sería una contradicción para la minimalidad del árbol de Steiner.

Observación 1.2. Sea s una hoja del árbol de Steiner y R el conjunto de vértices de ramificación del mismo. Si x es el elemento de $S \cup R$ más cercano a s en el árbol, con $x \neq s$, entonces debe cumplirse que

$$x \in N_{\delta(s, S\setminus\{s\})}(s)$$

Todos los vértices entre s y x son necesariamente de grado dos, o de lo contrario serían hojas o vértices de ramificación, por lo que una forma de interpretar esto es mirar a x como el punto donde s se conecta con el resto del árbol. Con este enfoque, el resultado nos dice que tal punto de conexión debe estar relativamente cerca. Tiene que encontrarse a lo más tan lejos como el vértice terminal más cercano a s que no sea s mismo.

Si no fuera así y x estuviera más lejos, se podría cambiar en el árbol la trayectoria de s a x por otra más corta que también conecte a s con los demás vértices terminales, lo cual resultaría en un árbol más pequeño. Esto no puede ocurrir, debido a la definición del árbol de Steiner.

Observación 1.3. Sea s una hoja del árbol de Steiner, y x el vértice adyacente a s. Obsérvese que si se poda s del árbol, entonces lo que queda es también un árbol de Steiner, pero para el conjunto de vértices $(S \setminus \{s\}) \cup \{x\}$ en G.

De lo contrario, habría un árbol más pequeño que contiene a $(S \setminus \{s\})$ y a x. Pegándole otra vez s, se obtendría un árbol para S más pequeño que el árbol de Steiner original.

Las dos últimas propiedades nos dicen mucho de la forma de los árboles de Steiner. Por ejemplo, que al buscar la solución, los vértices terminales tratan de conectarse entre si lo más rápido que pueden; que si se tienen muchos elementos de S muy cerca unos de otros, es probable que se generen varios vértices de

ramificación en el árbol en esa zona; que en general no puede haber muchas ramas muy largas, sino que se da preferencia a las ramas cortas.

La última propiedad también sugiere que si conociéramos algunos de los puntos de ramificación de antemano y pudiéramos encontrar de alguna manera un árbol de Steiner para ellos, quizá podríamos extenderlo a un árbol de Steiner para S.

Todas estas ideas son muy intuitivas. Más adelante formalizaremos algunas de ellas.

1.3. Un Algoritmo Codicioso

Las observaciones de la sección anterior sugieren que busquemos algo parecido a un árbol generador de peso mínimo, pero restringido de alguna manera solo a los vértices terminales.

La relación entre los dos tipos de árbol no es muy clara, ya que en general un árbol de Steiner no contiene a todos los puntos de la gráfica. Sin embargo, la experiencia de la Geometría Computacional en este tipo de problema nos ha enseñado que suele ser una buena opción intentar desarrollar un algoritmo codicioso. La idea es ir construyendo el árbol poco a poco, en cada paso eligiendo la pieza que, al añadirla, nos acerque lo más posible a la meta final (de aquí viene el nombre que se le da a este tipo de algoritmos). El problema con este enfoque es que al hacer esto existe el riesgo de que cerremos las puertas a la solución real: no siempre la solución que parece la mejor a corto plazo lo sigue siendo a largo plazo.

Aun así, muchos de los algoritmos que encuentran árboles generadores de peso mínimo trabajan (y funcionan) usando esta técnica codiciosa. Las ideas detrás de ellos son lo que motivan el algoritmo que describimos a continuación. Por ahora nos limitamos a describirlo; en capítulos posteriores estudiaremos formalmente sus propiedades.

Lo que haremos será fijarnos en las distancias en G que hay entre parejas de elementos de S, conectar a los dos que estén más cerca uno del otro, luego a los dos siguientes que estén más cerca (cuidando de no introducir ciclos), y así sucesivamente hasta que todas las parejas de vértices hayan sido consideradas. Originalmente, todos los vértices terminales están separados. Cada vez que conectamos dos el número de fragmentos disminuye. Eventualmente quedará un solo fragmento, i.e. una gráfica conexa. Como cuidamos de no introducir ciclos a cada paso, dicha gráfica debe ser un árbol.

Más formalmente, el algoritmo codicioso recibe una gráfica conexa $G = (V, E)$, y un subconjunto $S \subset V$. Sigue los pasos siguientes, en orden:

1. Para cada pareja de vértices $u, v \in S$ encontrar $\delta(u, v)$ y una trayectoria de esa longitud entre ellos.

2. Ordenar la lista de trayectorias, de las más cortas a las más largas. Inicializar la solución parcial H como una gráfica que contiene solo a los

elementos de S y ninguna arista. Hasta ahora, todos los vértices terminales están desconectados en H.

3. Tomar de la lista de trayectorias la más corta. Si los dos extremos de la trayectoria todavía están en componentes conexas distintas en H, la añadimos a la solución parcial. Si no, solamente la descartamos.

4. Si queda más de una componente conexa en S, repetimos desde el paso 3, si no ya terminamos.

El algoritmo es claro y bastante simple. Si lo analizamos con cuidado podemos ver que estamos construyendo una gráfica conexa (no paramos hasta que quede solo una componente conexa) y sigue nuestra intuición al conectar a los vértices terminales más cercanos primero.

1.4. Problemas del Enfoque Codicioso

Hay dos razones por las que la gráfica obtenida por el algoritmo de la sección anterior no necesariamente es un árbol de Steiner.

Primero, lo que el algoritmo termina construyendo no es necesariamente un árbol, pues no se eliminan todos los ciclos. Para ver por qué, analícese lo que pasaría si ejecutáramos el algoritmo en la gráfica de la figura 1.4. El problema surge debido a que algunas de las trayectorias que tomamos entre los vértices terminales pueden tener vértices interiores o aristas en común. Al juntar las trayectorias, es probable que algunas formen ciclos entre sí. El algoritmo no controla esto.

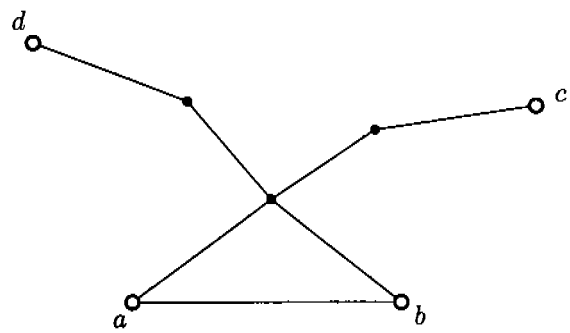


Figura 1.4: Al ejecutar el algoritmo codicioso sobre esta gráfica, el ciclo puede no ser eliminado. Por ejemplo, podría ser que se eligieran las trayectorias $a \rightsquigarrow b$, $b \rightsquigarrow c$ y $a \rightsquigarrow d$. Esta sería una elección válida, pero no se eliminaría ninguna arista.

Para resolver este problema podemos procesar la gráfica que devuelve el algoritmo cuando termina, con el objetivo de eliminar los ciclos que pueda tener. Una manera de hacerlo es encontrarle un árbol generador, usando alguno de

los algoritmos de recorrido de gráficas, por ejemplo BFS. Describiremos mas ampliamente este algoritmo en el capítulo 4.

El segundo problema es más serio. En general, no podemos ni siquiera extraer el árbol de Steiner de la gráfica que construye el algoritmo codicioso. Considérese la gráfica de la figura 1.5. Cada pareja de vértices blancos está a distancia 3, como se puede comprobar muy fácilmente a mano. Si tomamos 3 trayectorias de esa longitud, de forma que todos los vértices terminales queden conectados, habremos hecho lo que dice el algoritmo codicioso; dicha subgráfica contiene 10 vértices en total.

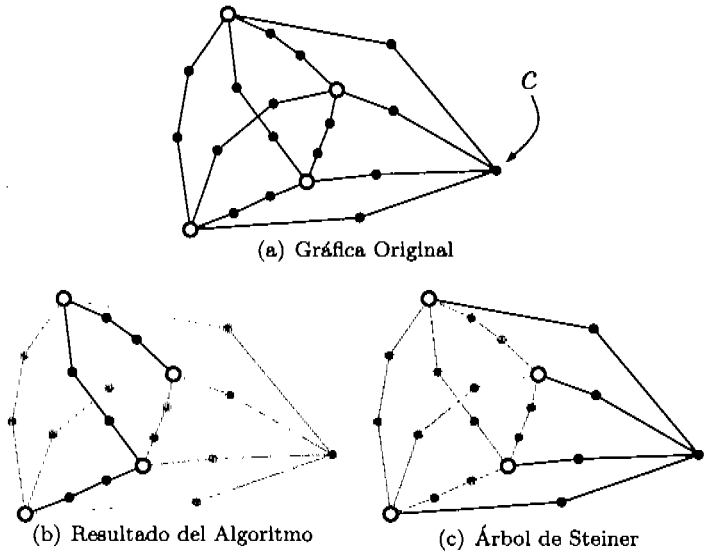


Figura 1.5: Una gráfica en la que el algoritmo codicioso falla.

Sin embargo, el árbol de Steiner para esta gráfica no contiene ninguna de esas aristas, sino que está formado de las trayectorias que van del vértice c a cada uno de los vértices terminales. Cada una de estas trayectorias tiene un vértice no-terminal distinto de c, y como son necesarias las 4 trayectorias, en total hay 9 vértices en este árbol. Si se juega un poco con la gráfica, es fácil convencerse que éste es el árbol más pequeño que contiene a todos los terminales. Por lo tanto es el árbol de Steiner.

La subgráfica que obtiene el algoritmo no solo está mal por tener más vértices que el árbol de Steiner, sino que en este caso las dos gráficas ni siquiera comparten una sola arista.

Una pregunta muy natural en este momento es, ¿qué tanto se puede alejar del árbol de Steiner el algoritmo codicioso? Para este ejemplo en particular no fue mucho, pues la gráfica encontrada por el algoritmo contiene solo 1 vértice más que el árbol de Steiner, pero no está claro lo que pasa en el caso general.

El ejemplo sirve para darnos cuenta por qué falla el algoritmo. Se trata

precisamente de su lado codicioso, que lo hace tomar la trayectoria más corta primero, cuando si tomara una un poco más larga podría de alguna manera acercar a varios otros vértices al mismo tiempo.

La construcción de la gráfica anterior se puede generalizar de la siguiente manera: tomamos n vértices blancos y trazamos entre cada pareja una trayectoria con 1 vértice negro nuevo, luego agregamos otro vértice negro al que llamamos c, y de éste ponemos una arista a cada vértice blanco. Los vértices blancos son los terminales, y se encuentran a distancia 2 unos de otros. Es perfectamente posible que el algoritmo escoja solo trayectorias que no pasan por c para conectarlos. Similarmente a como ocurrió antes, el árbol de Steiner consiste solamente del vértice c, y las aristas entre éste y los vértices blancos.

De las trayectorias que no pasan por c, se necesitan exactamente $n-1$ para conectar los n vértices terminales, luego el algoritmo podría darnos una gráfica con $2n-1$ vértices (n terminales y $n-1$ no-terminales), mientras que el árbol de Steiner tiene $n+1$ vértices. Una observación interesante es que la fracción

$$\frac{2n-1}{n+1} < 2$$

tiende a 2 cuando n crece. Es decir, que al menos para este tipo de gráfica, el algoritmo codicioso construye una subgráfica con un poco menos del doble de vértices que el árbol de Steiner en el peor caso.

Dado que nuestro primer algoritmo falló, podríamos tratar de corregirlo o proponer otro. Sin embargo, en el siguiente capítulo mostraremos que este problema es muy difícil de resolver, quizá incluso imposible. En ese caso, lo mejor que podemos hacer es buscar una buena aproximación al árbol de Steiner.

Capítulo 2

Problemas NP-Completos

En este capítulo estudiaremos las dificultades primordiales al tratar de resolver el problema del Árbol de Steiner.

En el capítulo 3 trataremos a fondo la relación profunda que existe entre este problema y el de buscar el Árbol Generador de Peso Mínimo de una gráfica. El hecho de que el algoritmo codicioso del capítulo 1 haya fallado, aún cuando sabemos que el mismo tipo de algoritmo funciona tan bien para encontrar árboles de peso mínimo, nos lleva a pensar que encontrar el árbol de Steiner es un problema mucho más difícil de resolver. Es decir, se trata de un problema NP-completo.

En la primera sección hablaremos un poco sobre la teoría de la complejidad y sobre los problemas NP-completos en general.

En la segunda definiremos formalmente el problema del Árbol de Steiner y mostraremos que es NP-completo.

Concluiremos el capítulo en la tercera sección, hablando un poco más de los Árboles de Steiner, las principales variantes del problema y su relación con nuestro estudio.

Para una exposición más detallada de los conceptos de complejidad introducidos en este capítulo, se recomienda revisar [4]. Los conceptos de algoritmos no deterministas, reducciones y NP-completez pueden ser difíciles de entender en un principio, por lo que se recomienda [6] como libro de referencia básica para estos temas.

2.1. Introducción a la Complejidad

Los algoritmos se pueden clasificar de acuerdo al tiempo que tardan en terminar de ejecutarse. En este caso, la unidad de tiempo que se utiliza por lo general no son segundos, sino que se cuenta el número de operaciones elementales que el algoritmo debe realizar antes de finalizar.

Lo que esta clasificación nos da es una relación entre el *tamaño* de la entrada del algoritmo y la complejidad del cálculo de la respuesta. Así, por ejemplo, po-

demos tener un algoritmo que necesite realizar un máximo de $20n^2$ operaciones elementales antes de poder regresar ordenado un conjunto de n números.

Dado que ejecutar una operación en dos máquinas diferentes por lo general toma tiempos diferentes, en general no nos preocupamos de clasificar los algoritmos con tanta exactitud. Simplemente decimos que el algoritmo anterior tarda a lo más tiempo cuadrático en el peor caso. Esto lo escribimos así: la *complejidad* del algoritmo anterior es $O(n^2)$.

Es muy posible que no todas las entradas del mismo tamaño se lleven el mismo tiempo en ser procesadas. El fijarnos en el peor caso nos permite saber qué es lo menos que podemos esperar del algoritmo y en base a esto hacer un análisis más seguro.

Otra razón por la que eliminamos las constantes al hacer la clasificación es que lo que queremos es darnos una idea de qué tan rápido aumenta la complejidad del algoritmo respecto al tamaño de su entrada. Aunque la constante que queda oculta puede ser muy grande, eventualmente el término variable es el que va a marcar la velocidad de crecimiento.

Entonces, al decir que un algoritmo tiene complejidad $O(f(n))$, lo que queremos decir es que conforme crece la entrada del algoritmo, el tiempo que este toma en terminar crece a lo más tan rápido como la función $f(n)$, salvo quizá por algunas constantes.

Debido a las limitantes físicas de las computadoras, los únicos algoritmos que son realmente útiles en la práctica son aquellos cuya complejidad se puede especificar con una función polinomial. Es decir, aquellos que en la clasificación anterior pertenecen a $O(n^k)$ para algun número natural k. Aquellas funciones que no se pueden acotar por alguna función polinomial, por ejemplo las exponenciales, simplemente crecen demasiado rápido como para ser útiles salvo con entradas muy pequeñas.

Para ejemplificar esto, supongamos que tenemos otro algoritmo para ordenar números que trabaja de la siguiente manera: genera todas las posibles permutaciones del conjunto de números y las va analizando hasta encontrar una que haya quedado ordenada. Para n números hay $n!$ permutaciones distintas, y en el peor caso se tienen que considerar todas. Incluso suponiendo que al implementar este algoritmo se pueden mantener las constantes muy bajas, por ejemplo que en total se tuvieran que hacer $\frac{n!}{100}$ operaciones elementales, para ordenar 10 números se tendrían que realizar hasta 36288 operaciones. Compárese esto con el algoritmo anterior, que necesitaría solamente 2000 operaciones elementales en el peor caso. La función factorial no se puede acotar con ninguna función polinomial.

2.1.1. P y NP

Aquellos problemas para los que podamos encontrar un algoritmo que los resuelva en tiempo polinomial, los ubicaremos en una clase a la que llamamos P. Estos son los problemas que, en la práctica, podemos esperar resolver en un tiempo razonable.

Similarmente, existe otra clase de problemas llamada NP[1]. En esta clase ubicamos a los problemas para los cuales podemos encontrar un algoritmo no-determinista que los resuelva en tiempo polinomial. Dado que las computadoras actuales son todas deterministas, este tipo de algoritmos no siempre son útiles para resolver problemas reales. Sin embargo, la importancia de esta clase no es solo teórica, pues a veces la podemos usar para darnos una idea de cuándo un problema puede ser muy costoso de resolver (en tiempo, principalmente).

Como los algoritmos deterministas también se pueden catalogar como no-deterministas, es muy claro que $P \subset NP$ (ver figura 2.1). Una de las preguntas más famosas en matemáticas es si la contención se da en el otro sentido, es decir, si $P = NP$. Hasta ahora no se ha encontrado una respuesta a esta pregunta, pero ciertamente se han realizado muchos esfuerzos por contestarla. Como consecuencia, se ha dado un gran impulso al desarrollo de la Teoría de la Complejidad.

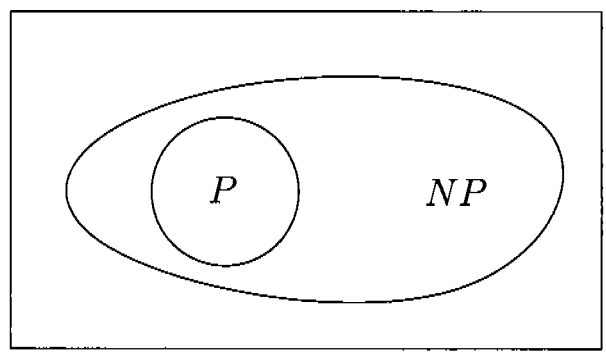


Figura 2.1: Las clases de complejidad P y NP no son opuestas, como a veces se cree. $P \subset NP$. ¿Será cierto que $P = NP$?

Un concepto muy importante que se ha creado a raíz de esto es el de los problemas *NP-completos*. Los problemas de este tipo tienen la sorprendente propiedad de que si alguien encuentra un algoritmo determinista de tiempo polinomial que resuelva alguno de ellos, cualquier otro problema de NP se puede resolver en tiempo polinomial a partir de dicho algoritmo. Encontrar un algoritmo así implicaría inmediatamente que $NP \subset P$ y que por lo tanto ambas clases de complejidad son realmente la misma.

Si se demuestra que un problema es NP-completo, es muy probable que sea imposible solucionarlo eficientemente en la práctica. La razón es que mostrar que un problema NP-completo está en P, significaría responder afirmativamente el acertijo más grande de la Teoría de la Complejidad, el cual ha mostrado no ser fácil de resolver.

Además, si resultara ser que las dos clases son distintas, se seguiría como resultado que ningún problema NP-completo está en P.

[1]El nombre viene del inglés: Non-deterministic Polinomial.

2.1.2. Reducciones

Las reducciones son una herramienta muy importante para demostrar que un problema es NP-completo. Una *reducción* es una transformación de instancias de un problema A en instancias de otro problema B, de forma que si resolvemos el segundo, podemos obtener también una solución para el primero. En este caso decimos que A se reduce a B. Intuitivamente, esto quiere decir que el problema B tiene que ser al menos tan "difícil" de resolver como el problema A, siempre que la reducción misma no sea muy complicada o costosa.

Observación 2.1. Supongamos que A es un problema, y que no sabemos si está en P o no. Si $B \in P$ es otro problema y podemos encontrar una reducción de A a B que se pueda describir como un algoritmo determinista de tiempo polinomial, podemos construir un algoritmo para mostrar que $A \in P$ también.

Esto es, para cada instancia de A la convertimos a una de B usando la reducción, luego resolvemos ésta con un algoritmo determinista polinomial para B, y finalmente recuperamos la solución para la instancia de A original. Este proceso conforma un algoritmo determinista polinomial y resuelve A. Esta observacion motiva la definición formal de los problemas NP-completos.

Definición 2.1. Un problema B es *NP-completo* si está en NP y para cualquier problema $A \in NP$ podemos encontrar una reducción polinomial determinista de A a B.

Observación 2.2. Si A es NP-completo, y se quiere probar que B también lo es, es suficiente con encontrar una reducción apropiada de A a B.

En otras palabras, no necesitamos encontrar una reducción a B para cada problema NP. Con encontrar una reducción de *algún* problema NP-completo a B, se puede concluir que B es NP-completo.

2.2. El Problema ST es NP-Completo

Denotaremos el problema de los Árboles de Steiner como ST (por sus siglas en inglés). En esta sección cambiaremos el enfoque que le damos al problema para poderlo manejar más fácilmente. Ahora lo trataremos como un problema de decisión, para el cual se requiere contestar solamente "sí" o "no". El problema **ST** es entonces el siguiente: dada una gráfica $G = (V, E)$ conexa, un subconjunto $S \subset V$, y un número natural k, ¿existe alguna subgráfica conexa H de G con a lo más k vértices, que contenga a todos los elementos de S?

Los algoritmos no-deterministas son peculiares en el sentido de que no especifican un mecanismo mediante el cual obtener la respuesta correcta. Describen lo que hay que hacer, pero no cómo. El esquema que siguen la mayoría es asi: obtener un objeto que pueda ser la solución y determinar si dicho objeto es o no la solución. Por ejemplo, en el problema de ordenar un conjunto de números, un algoritmo no-determinista se podría definir mediante dos pasos: primero obtener

un listado de números; segundo revisar que los números que están en el listado sean los mismos que los que queremos ordenar y que cada pareja consecutiva de números esté ordenada. Si se cumplen estas dos condiciones habremos encontrado una solución, sin embargo el algoritmo no especifica cómo se debe obtener el listado de números. Este algoritmo *resuelve* el problema desde el punto de vista de que cabe la posibilidad de que alguien genere un listado que sea la solución buscada y el algoritmo lo identifique. Puesto de otra manera, no podemos decir que el algoritmo está mal, ya que es posible que el listado que se construye en el primer paso sea la solución.

Para demostrar el resultado principal de este capítulo, requerimos probar dos cosas. La primera es que ST está en NP, es decir que hay un algoritmo no-determinista de tiempo polinomial que lo resuelve. La segunda es que hay un problema NP-completo reducible a ST.

Teorema 2.1. *$ST \in NP$*

Demostración. Sea $G = (V, E)$ una gráfica conexa, k un número natural y $S \subset V$. Lo primero que hay que hacer es obtener una subgráfica H de G. Aquí entra el no-determinismo, pero para probar que se puede realizar en tiempo polinomial, una manera de hacerlo es elegir algunas de las aristas de G y listarlas todas (junto con sus vértices) como la subgráfica H. Podemos realizar esto en tiempo $O(|E|)$.

Lo que falta ahora es un mecanismo para saber, en tiempo polinomial, si una subgráfica cualquiera contiene a todos los elementos de S y es conexa. Para esto conviene recorrer la subgráfica usando un algoritmo como DFS ([4]), que encuentra las componentes conexas de la gráfica sobre la que trabaja. DFS se puede ejecutar en tiempo $O(|V_H| + |E_H|)$, donde V_H y E_H son respectivamente los conjuntos de vértices y aristas de H. Al hacer el recorrido se van marcando aquellos vértices con los que el algoritmo se encuentre. Si al terminar quedó algún elemento de S sin marcar, se sigue que la subgráfica no contiene a todo S. Esta última revisión se puede llevar a cabo en tiempo $O(|S|)$. En total, y para simplificar un poco, como $V_H \subset V$, $E_H \subset E$ y $S \subset V$, toma $O(|V| + |E|)$ pasos verificar que una subgráfica cumple con los requisitos necesarios.

El algoritmo no-determinista utiliza el método descrito en el párrafo anterior para revisar que la subgráfica H que se construyó sea conexa y contenga a S. De ser así, cuenta el número de vértices que tiene, y si este número es menor o igual a k regresa "sí". En otro caso regresa "no". Todo lleva un total de $O(|V| + |E|)$. □

2.2.1. Cubierta Mínima de Vértices de una Gráfica

El problema de la Cubierta Mínima de Vértices, o MVC por sus siglas en inglés, es NP-completo ([6]). De hecho, es uno de los primeros problemas para los que se demostró ésto, y se puede describir como sigue: para una gráfica $G = (V, E)$ dada, encontrar un subconjunto $C \subset V$ mínimo en número de vértices tal que cada elemento de E tenga al menos un extremo en C.

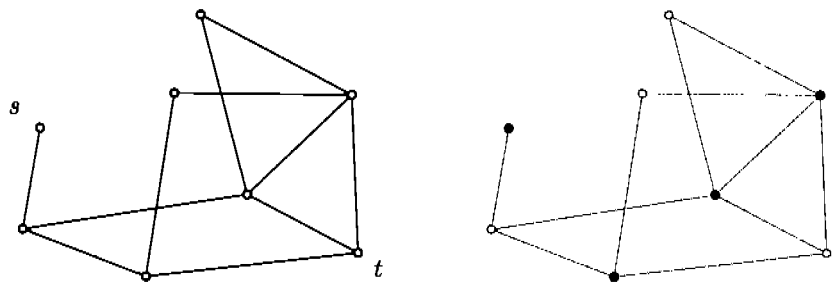


Figura 2.2: Una gráfica y una de sus posibles cubiertas mínimas de vértices. Los vértices negros forman la cubierta. Todas las aristas de la gráfica tienen al menos un extremo en la cubierta.

A cualquier subconjunto de vértices con esta propiedad se le llama *cubierta de vértices*, porque todas las aristas de la gráfica tocan alguno de los elementos del subconjunto. En la figura 2.2 mostramos una gráfica y su cubierta mínima.[2]

El concepto de cubierta nos permite definir MVC como un problema de decisión. Dada una gráfica G y un número natural k, ¿existe una cubierta de vértices para G con k elementos o menos?

Teorema 2.2. *MVC se puede reducir polinomialmente a ST.*

Demostración. Sea $G = (V, E)$ una gráfica y k un número natural. Construímos una nueva gráfica a partir de G en la cual resolver el problema ST implica a su vez una solución para MVC en G.

Primero se colorean todos los vértices de G con negro. A continuación, cada una de las aristas de E es reemplazada por dos aristas unidas por un vértice nuevo de color blanco, como se muestra en la figura 2.3. Si $e_i \in E$ es una arista, al vértice blanco que se genera al romperla le llamaremos b_i.

Por último se añade otro vértice blanco, el cual es unido con nuevas aristas a todos los vértices negros. A éste vértice le llamaremos o. Esto completa la transformación.

El proceso completo se ejemplifica en la figura 2.4. Nótese que la gráfica que resulta es conexa, sin importar si G lo era o no. A esta gráfica le llamaremos G_{ST}.

Observación 2.3. En G_{ST} la única manera de llegar de un vértice blanco a otro es pasando por alguno de los vértices de la gráfica original.

[2] Para ver que en efecto es una cubierta mínima, considérese el hecho de que para cualquier cubierta, si se tiene una trayectoria con l aristas en la gráfica, entonces la trayectoria debe contener al menos $\lceil l/2 \rceil$ vértices de la cubierta en su interior. Entonces, con el número de aristas de la trayectoria más larga de la gráfica, podemos obtener una cota inferior para el número de vértices de la cubierta mínima. En el ejemplo, existe una trayectoria que empieza en s y termina en t compuesta de 7 aristas. Se sigue que la cubierta mínima debe tener por lo menos 4 vértices. Se puede comprobar manualmente que el conjunto de vértices negros en la figura forman una cubierta.

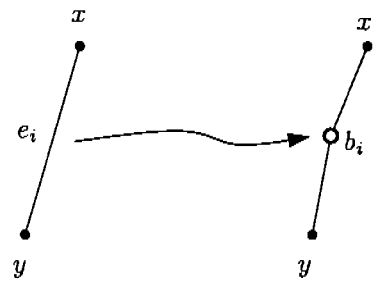


Figura 2.3: Cada arista es reemplazada por otras dos unidas por un vértice blanco.

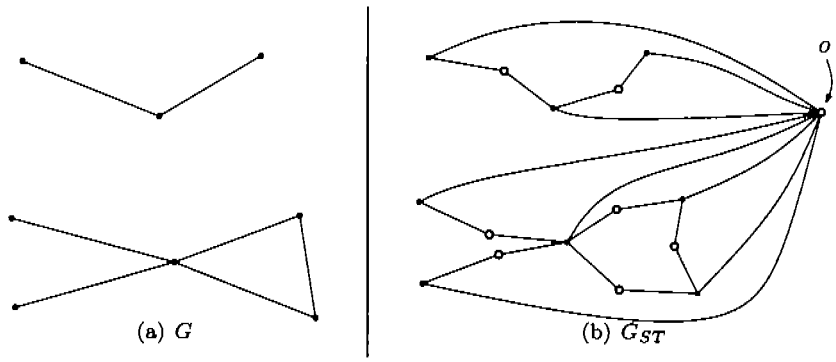


Figura 2.4: (a) Una gráfica y (b) el resultado de aplicarle la reducción.

Ahora supóngase que se obtiene un árbol de Steiner para los vértices blancos de G_{ST}. Como dicho árbol debe contener a todos estos vértices, lo único que puede minimizar es el número de vértices negros que requiere. Pero ¿cuántos vértices negros se necesitan?

Lema 2.1. *Si C es el conjunto de vértices negros que aparece en el árbol de Steiner de G_{ST}, C forma una cubierta mínima de vértices para G.*

Demostración. Sea $e_i = (x, y) \in E$. Tenemos que b_i es adyacente a x y y en G_{ST}. Por la observación 2.3 x o y deben estar en C. Así, si $e \in E$, entonces uno de sus extremos está en C. Esto es, C es una cubierta de vértices para G.

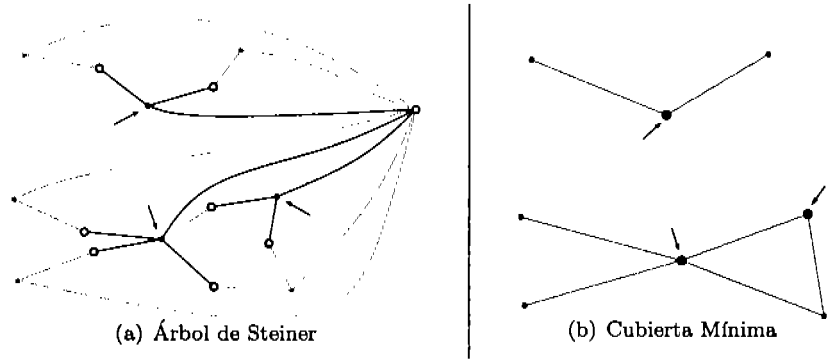
Ahora nos fijaremos en una cubierta cualquiera de G y la utilizaremos para construir una gráfica en G_{ST} con al menos tantos vértices negros como el árbol de Steiner, de donde se seguirá la minimalidad de C.

Supongamos que tenemos una cubierta de vértices de G. A cada b_i lo conectamos con el extremo de e_i que pertenezca a la cubierta (si sus dos extremos pertenecen a la cubierta, elegimos arbitrariamente uno de ellos). Agregamos también las aristas entre los vértices de la cubierta y el vértice o. Sea H la subgráfica de G_{ST} que resulta de esta construcción.

Tomar todas las aristas entre o y los vértices de la cubierta, nos asegura que H es conexa, y por construcción contiene a todos los vértices blancos de G_{ST}. Por definición debe tener al menos tantos vértices como el árbol de Steiner para B. Luego, el número de vértices negros en H debe ser mayor o igual al número de vértices negros en el árbol de Steiner.

Como C es el conjunto de vértices negros de un árbol de Steiner, si γ es el número de vértices de una cubierta mínima de vértices de G lo que acabamos de demostrar es que $\gamma \geq |C|$, lo cual prueba la minimalidad de C. Esto termina la demostración del lema. □

Este lema implica que si resolvemos ST para G_{ST}, es decir, si encontramos que el árbol de Steiner de G_{ST}, con los $|E|+1$ vértices blancos como terminales, contiene a lo más $k + (|E| + 1)$ vértices, habremos resuelto también MVC para G y k (véase la figura 2.5).



(a) Árbol de Steiner (b) Cubierta Mínima

Figura 2.5: Los vértices negros que aparecen en un árbol de Steiner de la gráfica modificada inducen una cubierta mínima de vértices en la gráfica original.

Utilizando una representación de gráficas con matrices de adyacencia podemos colorear los vértices de G en tiempo $O(|V|)$, romper las aristas en $O(|E|)$, y añadir el vértice o en $O(|V|)$. Por lo tanto, la reducción se puede llevar a cabo en tiempo polinomial respecto a G. □

Corolario 2.1. *ST es NP-completo.*

Demostración. El problema está en NP, por el teorema 2.1. Ahora bien, sea $A \in NP$ un problema. Como MVC es NP-completo, se puede reducir polinomialmente A a MVC. Si después se aplica la reducción del teorema 2.2 de MVC a ST, este proceso resulta en una reducción polinomial de A a ST. Por lo tanto ST es NP-completo. □

2.3. Árboles de Steiner

Este problema es uno de los más famosos en la Geometría Computacional y tiene múltiples aplicaciones. En su forma más general, dada una gráfica conexa

$G = (V, E, w)$ con pesos en las aristas y un suconjunto $S \subset V$ de vértices terminales, hay que encontrar una subgráfica conexa de G, mínima en peso, que contenga a S (aquí abusamos un poco de la terminología, cuando nos referimos al peso de una gráfica como la suma de los pesos de sus aristas).

Cuando los pesos son todos positivos, la subgráfica que se busca es un árbol, pues si contuviera ciclos se podría eliminar alguna arista sin perder la conexidad, reduciendo el peso total de la subgráfica.

Hay algunos casos particulares de esta definición que son importantes por sí mismos. Por ejemplo, cuando V consiste de los puntos del plano, E son todas las aristas entre ellos, y w es la distancia euclidiana, se le conoce como Problema del Árbol de Steiner Euclideano. De aquí viene la parte de "Steiner" del nombre:[3] al estudiar este problema, da la impresión que lo que se hace es buscar nuevos puntos en el plano para agregarlos al conjunto de vértices terminales.

Cuando en vez de la distancia euclidiana se utiliza la distancia Manhattan, al problema se le llama Árbol de Steiner Rectilíneo. Estas dos variantes son muy importantes en el diseño de redes de todo tipo, en la construcción de circuitos de computadora, y en general en cualquier problema que requiera conectar puntos en un plano con el menor costo posible.

Es un resultado conocido de la Teoría de Gráficas que el número de aristas de un árbol es siempre uno menos que su número de vértices. Por esta razón en una gráfica en la que el peso de todas las aristas es el mismo, al buscar el Árbol de Steiner se minimizan al mismo tiempo el número de vértices y el de aristas. Con este enfoque podemos ver que el problema que hemos estado estudiando hasta ahora es realmente un caso particular del problema del Árbol de Steiner, en el que $w(e) = 1$ para todo elemento e de E.

El problema general del Árbol de Steiner es uno muy estudiado y está entre los primeros problemas NP-completos que se encontraron ([8]). El resultado de que nuestra variante es NP-completa, a pesar de tratarse de un caso particular, es un conocido.

Seguiremos refiriéndonos al problema restringido como hasta ahora. Es decir, para nosotros el árbol de Steiner es aquel que minimiza el número de aristas, sin importar el peso que puedan tener. En los casos en que los resultados se apliquen también a otras formas del problema general del Árbol de Steiner, lo mencionaremos explícitamente.

[3]En Geometría Computacional, los Puntos de Steiner son aquellos que no pertenecen al problema original, pero que se agregan para poder encontrar una aproximación más fácilmente.

Capítulo 3

Aproximaciones y Soluciones

Los resultados del capítulo 2 sugieren que encontrar una solución al problema del Árbol de Steiner es extremadamente difícil, quizá incluso imposible. Si en vez de resolverlo encontramos alguna manera de aproximar la solución eficientemente, habremos hecho un avance importante. En este capítulo mostraremos como se puede hacer una buena aproximación usando las ideas presentadas anteriormente.

En la primera sección estudiaremos al algoritmo codicioso presentado en el capítulo 1 con mucha más atención, así como su relación con el problema del Árbol Generador de Peso Mínimo.

En la segunda sección veremos que nuestro algoritmo construye en realidad una 2-aproximación al Árbol de Steiner. Si consideramos que el problema es NP-completo, esta resulta ser una buena aproximación. También discutiremos por qué las ideas que usamos fallan, aun cuando funcionan tan bien para otros problemas similares.

La sección 3 profundiza un poco más en esto y presenta una heurística que en muchos casos nos permite aproximar mejor el árbol de Steiner. Introducimos el concepto de conjunto fundamental de un árbol de Steiner, mediante el cual se puede reconstruir eficientemente el árbol completo usando la gráfica original.

En la sección 4 estudiamos un problema ligeramente más restringido, en el cual el número de vértices terminales no puede exceder cierta cantidad. Cuando se cumple esta condición extra, siempre podemos encontrar el árbol de Steiner en tiempo polinomial.

Concluímos el capítulo discutiendo los distintos resultados mostrados hasta el momento, así como su importancia.

3.1. ST y MST

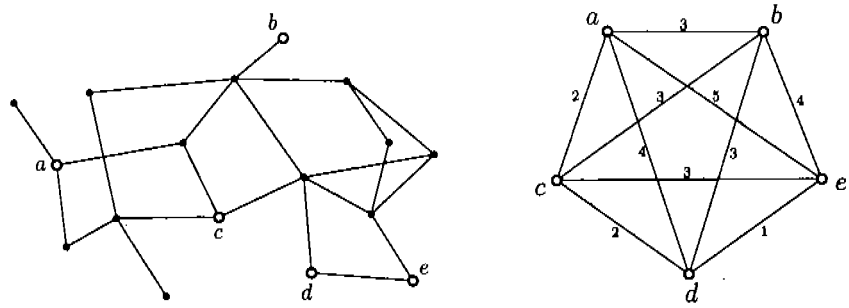
El problema del Árbol Generador de Peso Mínimo, o MST por sus siglas en inglés, es fundamental en el estudio de algoritmos de geometría computacional. En términos generales, para una gráfica conexa dada lo que se busca es la subgráfica conexa más ligera que contenga a todos los vértices. Cuando los pesos de las aristas de la gráfica son todos positivos, la subgráfica resulta ser un árbol. Como ya se habrá notado, se trata de un caso particular del problema general del Árbol de Steiner, en el cual todos los vértices son terminales.

Prácticamente todos los algoritmos que resuelven este problema se basan en la idea de construir el árbol incrementalmente. En cada paso seleccionan una arista *segura* (la más ligera, que no genere un ciclo), la añaden y repiten hasta que todas las aristas hayan sido consideradas. Esta estrategia *codiciosa*, que en cada paso selecciona la opción que más le favorece, es la misma que utilizamos en el algoritmo del capítulo 1.

La siguiente definición nos permite relacionar aun más los conceptos de Árbol Generador de Peso Mínimo y Árbol de Steiner.

Definición 3.1. Sea $G = (V, E)$ una gráfica conexa y $S \subset V$ un subconjunto de sus vértices. Sean $D = \{(a, b) | a, b \in S\}$ y $d : D \to \mathbb{R}$, definida como $d(a, b) = \delta_G(a, b)$. La *Gráfica de Proximidad* asociada a G y S es (S, D, d), y la representaremos como $GP_G(S)$.

Con la gráfica de proximidad representamos la distancia en G que existe entre cada par de elementos del conjunto S. En la figura 3.1 se ejemplifica este concepto.



(a) G (b) $GP_G(S)$

Figura 3.1: Un ejemplo de gráfica de proximidad. El peso de cada arista (x, y) de $GP_G(S)$ corresponde a la distancia que hay en G entre sus x y y. El conjunto S está dado por los vértices blancos en (a).

Sea $G = (V, E)$ conexa y $S \subset V$. Notemos que el algoritmo codicioso que propusimos en el capítulo 1 se puede reescribir de la siguiente manera:

1. Encontrar $GP_G(S)$ y ordenar sus aristas en orden creciente por peso. Iremos construyendo una subgráfica arista por arista. Inicialmente toma-

mos todos los vértices de $GP_G(S)$ pero ninguna arista, de forma que esta subgráfica esté totalmente desconectada.

2. Si la gráfica construída hasta ahora consta de una sola componente conexa, el algoritmo termina. En otro caso se ejecuta el siguiente paso y al terminar repite desde aquí.

3. Considerar a la arista más ligera que aun no haya sido procesada. Añadirla solo si sus dos extremos pertenecen a componentes conexas diferentes en la solución parcial. Si no, descartarla.

El paso 3 del algoritmo es el método de Kruskal para obtener un árbol generador de peso mínimo de una gráfica. En otras palabras, el algoritmo codicioso primero construye la gráfica de proximidad de los vértices terminales y luego le encuentra un árbol generador de peso mínimo.

La transformación que hicimos del algoritmo no es precisa. La versión del capítulo 1 construye una subgráfica conexa que contiene a todos los vértices terminales *en* G, mientras que el árbol que construimos aquí es una subgráfica de $GP_G(S)$. Sin embargo, podemos usar muy fácilmente este árbol para reconstruir la solución en G, pues cada una de las aristas (u, v) del árbol corresponde a una trayectoria de peso mínimo entre u y v en G. El proceso consiste en tomar a la subgráfica H formada por todas estas trayectorias juntas. Como el árbol en $GP_G(S)$ es una gráfica conexa que contiene a los elementos de S, H también lo es. El único problema es que al juntar las trayectorias se pueden generar ciclos, por lo tanto necesitamos encontrar un árbol generador en H (cualquier árbol; BFS o DFS funcionan bien para esto) para que la transformación sea completa.

Observación 3.1. El peso del árbol generador de peso mínimo de la gráfica de proximidad proporciona una cota superior al número de aristas que tiene el árbol de Steiner.

3.2. Una 2-Aproximación

Nuestro algoritmo codicioso falla en encontrar el árbol de Steiner. Sin embargo, la gráfica que construye es una aproximación bastante buena.

Teorema 3.1. *La gráfica que se obtiene por el algoritmo codicioso tiene a lo más el doble de aristas que el árbol de Steiner.*

Demostración. Sea $G = (V, E)$ una gráfica conexa, $S \subset V$, y T un árbol de Steiner para S. Consideraremos a T como una gráfica plana, es decir, dibujada en el plano sin formar cruces entre las aristas.

Para el caso en que S contenga solo un elemento, el resultado es trivial. Cuando S contiene al menos 2 elementos, podemos ordenarlos usando el árbol con el método que describimos a continuación. Primero elegimos arbitrariamente un elemento de S y le llamamos s_0. Ahora comenzamos un recorrido desde s_0 manteniendo el árbol siempre a la derecha. Eventualmente encontraremos otro

vértice de S, y a éste le llamamos s_1. Continuamos este proceso hasta regresar al vértice s_0, con la observación de que si en algún momento del recorrido pasamos por un vértice que habíamos nombrado previamente, lo saltamos y seguimos.[1] Esto evita introducir duplicados en la lista final. Véase la figura 3.2 para un ejemplo. Si $|S| = n - 1$, por comodidad para manejar nuestra notación diremos que $s_n = s_0$.

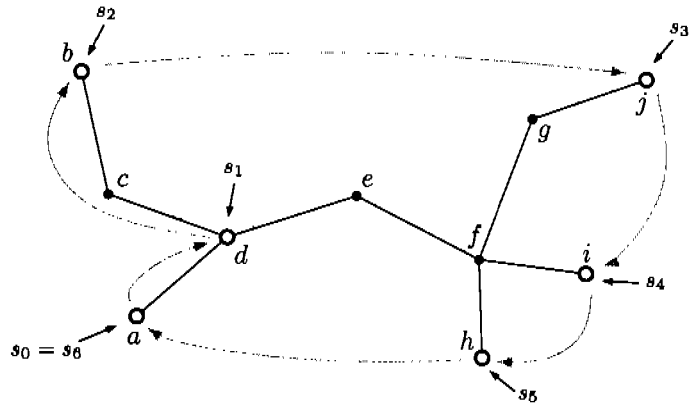


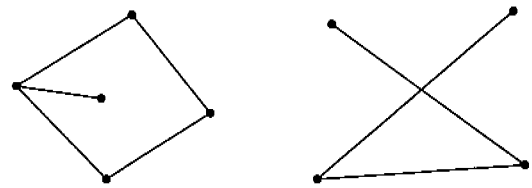
Figura 3.2: Se puede usar el dibujo de un árbol para establecer un orden entre sus vértices. En este caso, el recorrido comienza en el vértice a y sigue el sentido de las manecillas del reloj. Cuando ocurren repeticiones, se da preferencia a la primera aparición. El resultado en este caso es la sucesión $\{a, d, c, b, e, f, g, j, i, h\}$. Obsérvese que, por ejemplo, después de pasar por b no se vuelven a mencionar a c ni a d, sino que se salta directamente a e. El orden inducido para los vértices terminales está marcado por las etiquetas s_i en la figura.

Como no hay cruces de aristas, no hay ambigüedad en la manera de realizar el recorrido. Además, dado que T es una gráfica conexa, si al terminar existiera algún vértice sin tocar o alguna arista que no hubiera sido recorrida por los dos lados, tendría necesariamente que existir un ciclo que no nos dejara "dar la vuelta" para recorrer dicha parte de T. La figura 3.3 ilustra esta situación.

Pero T no contiene ciclos, por lo que después de realizar el recorrido debemos haber pasado por todos sus vértices al menos una vez, y por cada una de sus aristas exactamente 2 veces (una en cada dirección).[2] Podemos hablar del *peso* del recorrido para referirnos a la suma de los pesos de las aristas por las que

[1] Una manera de visualizar esto es imaginar que las aristas del árbol son muros, los vértices son torres, y nosotros estamos parados en el suelo junto a una de estas torres. El recorrido consiste en caminar manteniendo la muralla siempre a nuestra derecha, yendo de torre a torre. Obsérvese que cuando eventualmente regresemos a la torre inicial habremos recorrido cada muro por los dos lados, pasando al menos una vez por cada torre.

[2] Más formalmente, el recorrido se define independientemente de si hay o no ciclos, pero entonces hay que observar que lo que hacemos es recorrer una cara completa de la gráfica hasta que regresamos al vértice del que partimos. Cuando la gráfica es un árbol, como en nuestro caso, la única cara que existe es la exterior, por lo que terminamos recorriendo el árbol completo.



(a) (b)

Figura 3.3: (a) La única manera en que no podamos recorrer una arista por los dos lados es que no podamos "dar la vuelta" para llegar de un lado al otro de la misma. Esto es, que cada lado pertenezca a caras diferentes de la gráfica. Pero esto significa que tiene que haber un ciclo.
En (b) se muestra que la planaridad de la gráfica nos permite definir el recorrido más fácilmente. Si hubiera algún cruce, la instrucción de mantener el árbol siempre a la derecha perdería sentido, o bien haría que algunas aristas no se recorrieran completamente o por los dos lados.

pasamos, que por la observación anterior resulta ser $2 \cdot w(T)$, donde $w(T)$ es el peso del árbol. Por supuesto, si tomamos la distancia que hay en T de s_0 a s_1, y luego la que hay de s_1 a s_2, y asi sucesivamente hasta cerrar con s_{n-1} y s_n, la suma de todas ellas no puede valer más que el peso del recorrido. Entonces,

$$\sum_{i=0}^{n-1} \delta_T(s_i, s_{i+1}) \leq 2 \cdot w(T)$$

Como T es una subgráfica de G, $\delta_G(u,v) \leq \delta_T(u,v)$ para cualquier par de vértices $u, v \in T$. Luego,

$$\sum_{i=0}^{n-1} \delta_G(s_i, s_{i+1}) \leq 2 \cdot w(T)$$

Cada una de las distancias en G entre vértices consecutivos de S corresponde al peso de una arista de la gráfica de proximidad de S. Por lo tanto, si en $GP_G(S)$ nos fijamos en todas las aristas de la forma (s_i, s_{i+1}), con $0 \leq i \leq n-1$, obtendremos un ciclo C cuyo peso es a lo más dos veces el de T (figura 3.4). Le podemos quitar cualquier arista a este ciclo para obtener un árbol generador de $GP_G(S)$. Sea T' un árbol generador de peso mínimo de la gráfica de proximidad, entonces

$$w(T') < w(C) \leq 2 \cdot w(T)$$

Anteriormente, establecimos que lo que hace el algoritmo codicioso es calcular un árbol generador de peso mínimo en la gráfica de proximidad, por lo que esto completa la prueba. □

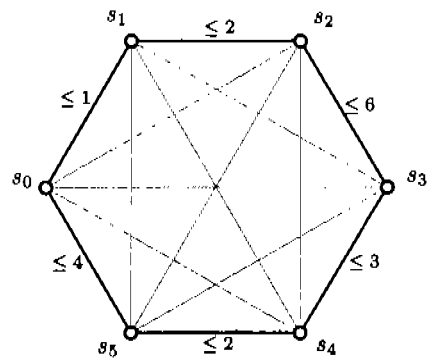


Figura 3.4: La gráfica de proximidad de la gráfica asociada al árbol de la figura 3.2. Las aristas oscuras representan el ciclo inducido por el ordenamiento de los vértices terminales. El peso de cada arista del ciclo está acotado por la distancia que hay entre sus extremos en el árbol. Nótese que $w(C) \leq 18 = 2 \cdot w(T)$, donde C es el ciclo y T es el árbol.

Juntando el resultado de esta sección con la observación 3.1, si T es el árbol de Steiner de S en G y T' el árbol generador de peso mínimo de $GP_G(S)$, hemos mostrado que

$$w(T) \leq w(T') < 2 \cdot w(T)$$

Observación 3.2. La cota del teorema anterior es justa.

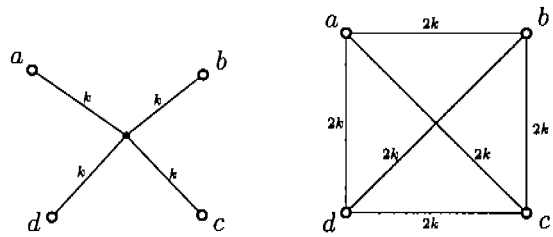
La familia de gráficas que presentamos en el capítulo 1 como contraejemplos a la correctez del algoritmo codicioso (véase la sección 1.4) nos sirven para establecer este punto. Esto es, hay ejemplos en los que el peso de T' se aproxima tanto como queramos al doble del peso de T.

La importancia de nuestra aproximación radica principalmente en que la cota es buena y el algoritmo es bastante simple. Analizaremos su implementación y complejidad en el capítulo 4. Otra razón de su importancia es que si hacemos algunas modificaciones podemos aplicar las mismas ideas para encontrar aproximaciones tan cercanas a la solución como queramos. Esto es lo que veremos a continuación.

3.3. Mejorando la Aproximación

La razón por la que el algoritmo falla es que las trayectorias que unen a los distintos vértices se pueden cruzar unas con otras arbitrariamente, incluso repitiendo aristas. En ocasiones, estos cruces pueden ayudar para acercar a los vértices unos con otros, en el sentido de que cada vértice terminal está más cerca del cruce que de alguno de sus similares. La abstracción hecha por la gráfica de proximidad no toma en cuenta los cruces, sino que se va directamente sobre las trayectorias más cortas entre cada pareja de vértices terminales. Las

trayectorias que son un poco más largas, pero que contienen cruces benéficos para la solución, ni siquiera son consideradas.



(a) (b)

Figura 3.5: (a) Una gráfica con un conjunto de vértices terminales marcado (los vértices blancos). Cada arista representa una trayectoria de longitud k. Su árbol de Steiner es la gráfica misma, cuyo peso es $4k$.
(b) La gráfica de proximidad asociada. En ella, cada arista tiene peso $2k$. Cualquier árbol Generador de Peso Mínimo requiere 3 aristas, por lo que su peso será $6k$. Esta información resulta engañosa porque no considera los cruces que hay entre las trayectorias de la gráfica.

La gráfica de la figura 3.5 ilustra muy claramente el problema. Las hojas son los vértices terminales, y se encuentran a distancia $2k$ unos de otros. Por la forma en que fue definida, la gráfica de proximidad asociada a esta estrella está desechando información muy útil; es decir, que cada hoja está a distancia k del vértice central. Este vértice es un cruce entre cualquier par de trayectorias entre hojas.

En una gráfica cualquiera, donde se pueden encontrar subgráficas similares a la de la figura anterior, el algoritmo dará preferencia a trayectorias más cortas entre los vértices terminales, ignorando completamente los cruces que los acercan.

Estas observaciones sugieren que podemos mejorar la aproximación si identificamos previamente los cruces importantes en la gráfica. Este no es un problema trivial, ya que prácticamente cualquier vértice de grado por lo menos 3 puede ser un cruce valioso. Aun más, un vértice de cruce que sea muy útil en un caso puede ya no serlo en otro caso ligeramente distinto.

Para atacar este problema hay que entender que los cruces corresponden a vértices de ramificación en el árbol. Si logramos identificar algunos elementos del conjunto de vértices de ramificación del árbol de Steiner, es posible que podamos usarlos para encontrar el árbol mismo. Esta es la idea fundamental detrás de la discusión de esta sección.

Definición 3.2. El *conjunto fundamental* de un árbol de Steiner T está compuesto por todos sus vértices terminales (ya sean hojas o vértices interiores), y sus vértices de ramificación. Lo denotaremos como $F(T)$.

Si conocemos $F(T)$ para algún árbol de Steiner T, el algoritmo codicioso

puede reconstruir un árbol de Steiner para el conjunto de vértices terminales original. Esto es útil, pues realmente no necesitamos a T, sino que nos basta con saber cual es su conjunto fundamental para poder recalcular un árbol de Steiner similar a el. Demostraremos esta propiedad a continuación.

Teorema 3.2. *Sea $G = (V, E)$ una gráfica conexa, $S \subset V$, y T un árbol de Steiner para S en G.*

Si S' es un subconjunto de vértices de T, tal que $F(T) \subset S'$, entonces la gráfica que construye el algoritmo codicioso a partir de S' es un árbol de Steiner para S.

Demostración. Supongamos que $u, v \in S'$ son dos vértices tales que la trayectoria que los une en T no contiene a ningún otro vértice de S'. Como T pesa lo menos posible, dicha trayectoria debe ser la más ligera que une a u y a v en G. Es decir, $\delta_T(u, v) = \delta_G(u, v)$.

Pero entonces la trayectoria entre u y v en el árbol induce una arista en la gráfica de proximidad asociada a G y S'. Mostraremos que el árbol completo se puede descomponer en trayectorias de este tipo,[3] de forma que lo que nos quede sea un árbol generador en $GP_G(S')$ con el mismo peso de T.

Hacemos la descomposición inductivamente sobre la cardinalidad de S'. Si S' tiene solo 2 vértices, la trayectoria que los une en T no puede contener ningún otro elemento de S'.

Si S' tiene 3 o más vértices, sea u una hoja y v el vértice de S' más cercano en T a u. Por las observaciones del capítulo 1, sabemos que $u \in S \subset S'$. Además, en la trayectoria que une a u y v no puede haber otro elemento de S', pues estaría más cerca de u que v. Los vértices interiores de esta trayectoria deben tener grado 2, pues S' contiene a todos los vértices de T de grado mayor o igual a 3 (los vértices de ramificación). Por lo tanto, si la quitamos obtenemos nuevamente un árbol, con todas sus hojas en el conjunto $S' - u$.

Por inducción podemos obtener una descomposición para el árbol reducido, y al juntarla con la trayectoria entre u y v queda una descomposición en trayectorias para T.

En esta descomposición, los extremos de cada trayectoria pertenecen a S' y los vértices interiores no, por lo que se pueden mapear directamente a aristas de $GP_G(S')$, como mencionamos antes. Cada vértice de S' está en al menos una trayectoria, por lo que el conjunto de aristas inducidas en la gráfica de proximidad constituye una subgráfica generadora H. Esta subgráfica hereda la conexidad de T, pues cada camino entre elementos de S' en T induce un camino entre los mismos elementos en H. Un argumento simétrico nos permite concluir que H no puede tener ciclos, por lo que se trata de un árbol. Finalmente, cada una de las aristas de T está en una y solo una trayectoria de la descomposición, por lo que el peso de H es el mismo que el peso de T. Véase la figura 3.6.

El algoritmo codicioso construye un Árbol Generador de Peso Mínimo para $GP_G(S')$ y de aquí extrae un árbol para S' en G, que por las observaciones

[3] Las trayectorias en la "descomposición" pueden compartir sus vértices extremos, por lo que realmente se trata de una descomposición de las *aristas* del árbol.

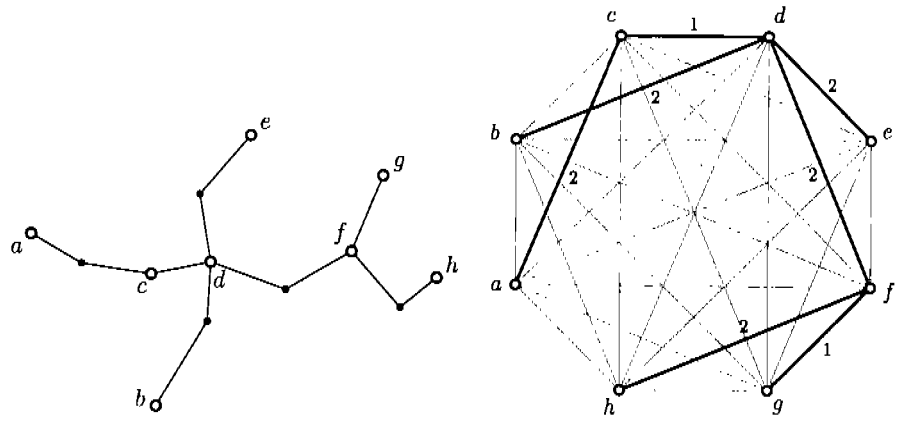


Figura 3.6: El árbol de Steiner de la izquierda contiene a su conjunto fundamental. Su descomposición en trayectorias induce un árbol generador en la gráfica de proximidad.

anteriores pesa a lo más tanto como T. Pero $S \subset F(T) \subset S'$, luego la gráfica que construye el algoritmo codicioso es un árbol de Steiner para S. □

Si logramos identificar los vértices de ramificación de algún árbol de Steiner, aunque no tengamos al árbol mismo, el teorema anterior nos dice que podemos usar el algoritmo codicioso para encontrar una solución exacta. En otras palabras, podemos reconstruir el árbol de Steiner completo (o uno similar) si sabemos cual es su conjunto fundamental.

La manera más simple de hacer esto es probar todos los posibles subconjuntos de vértices de la gráfica, tratarlos como si fueran los vértices de ramificación que buscamos, y usar el algoritmo codicioso con ellos. De todos los árboles que encontremos con esta técnica, aquellos que pesen lo mínimo serán árboles de Steiner para el conjunto original de vértices terminales.

Por supuesto esto no es viable computacionalmente, ya que el número de subconjuntos que se pueden hacer con n elementos es 2^n. Aun cuando pudiéramos implementar el algoritmo codicioso de una manera extremadamente eficiente, la complejidad del algoritmo que obtendríamos sería exponencial.

3.3.1. Un Algoritmo Codicioso Incremental

Un enfoque diferente es el de ir construyendo el conjunto fundamental poco a poco. Considérese la siguiente heurística.

Usamos el algoritmo codicioso para encontrar la aproximación inicial al árbol de Steiner partiendo solamente de los vértices terminales. Esta será la primera iteración del proceso. Ahora consideramos todos los árboles T_α que obtenemos con el algoritmo codicioso usando conjuntos de la forma $S \cup \{x_\alpha\}$, donde S es el conjunto de vértices terminales de la iteración anterior, y x_α es cualquier vértice de la gráfica que no esté en S. Sea T_{α_0} un árbol de peso mínimo de entre todos

los árboles que encontramos. Iteramos este proceso, tomando ahora a $S \cup \{x_{\alpha_0}\}$ como el nuevo conjunto de vértices terminales. Terminamos cuando el conjunto de árboles "extendidos" que obtengamos pesen todos lo mismo o más que el árbol elegido en la iteración anterior.

La idea detrás de este proceso es encontrar en cada paso el vértice que parezca el mejor candidato a pertenecer al conjunto fundamental del árbol de Steiner. Elegimos alguno que, al añadirlo, reduzca al mínimo el peso del árbol.

Como veremos en el capítulo 4, este *algoritmo codicioso incremental* se puede implementar de forma que su complejidad no sea mucho mayor a la del algoritmo codicioso. Además, dado que en cada iteración exitosa bajamos el peso del árbol encontrado, la aproximación que obtenemos con este método puede ser mejor que la del algoritmo codicioso.

Sin embargo no siempre podemos identificar los elementos del conjunto fundamental de un árbol de Steiner de esta manera. Por ejemplo, en la gráfica de la figura 3.7, usando esta técnica se construye un árbol distinto del Árbol de Steiner. De hecho no se logra identificar ni uno de los vértices de ramificación que nos interesan. La razón es que al considerarlos de uno en uno, los vértices de ramificación no bajan el peso total de la aproximación tanto como lo hace el vértice c. En este ejemplo, dicho vértice funciona como un "cebo" que atrae al algoritmo hacia una parte de la gráfica que no tiene nada que ver con el árbol de Steiner.

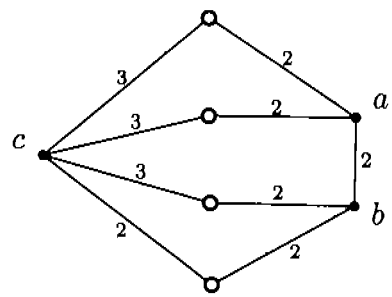


Figura 3.7: El árbol de Steiner de esta gráfica consiste de los vértices blancos junto con a, b, y todas las aristas entre ellos, y su peso es de 10. Sea w_x el peso del Árbol Generador de Peso Mínimo de la Gráfica de Proximidad que resulta al considerar a los vértices blancos junto con x como vértices terminales. Entonces $w_a = w_b = 12$, pero $w_c = 11$. Por lo tanto la heurística seleccionará a c como elemento del conjunto fundamental. Al hacer esto nos alejamos definitivamente del Árbol de Steiner real.

Aun así, en la práctica casi siempre podemos mejorar la aproximación del algoritmo codicioso usando esta estrategia. Dependiendo de que tanta complejidad extra se esté dispuesto a aceptar, se pueden añadir los elementos de dos en dos, o en bloques más grandes. Mientras más elementos agreguemos por iteración, más subconjuntos diferentes tendremos que considerar, y por lo tanto mayor será la complejidad del algoritmo codicioso incremental, pero la aproximación

obtenida será mejor en muchos casos.

3.4. Soluciones Exactas

En esta sección presentamos una colección de algoritmos que, cuando el número de vértices terminales no excede cierta cantidad, solucionan el problema del Árbol de Steiner. Esta versión del problema es diferente a la planteada en el capítulo 2, en donde probamos que el problema del árbol de Steiner para conjuntos arbitrarios de vértices terminales es NP-completo.

Para los resultados que presentamos a continuación, suponemos que el número de vértices terminales es fijo, o más precisamente, que no excede cierto número λ. No se trata del mismo problema. Con esta restricción extra podemos mostrar un algoritmo de tiempo polinomial que encuentra el árbol de Steiner exacto y no solo una aproximación.

3.4.1. Restringiendo el Espacio de Búsqueda

El siguiente lema es un resultado de teoría de gráficas. Nos permite restringir mucho el número de subconjuntos de vértices de G que hay que considerar como posibles candidatos a vértices de ramificación del árbol de Steiner.

Lema 3.1. *Un árbol con t hojas puede tener a lo más $t-2$ vértices de ramificación.*

Demostración. Sea $T = (V, E)$ un árbol. Sea t el número de hojas de T, s su número de vértices de grado 2 y r su número de vértices de ramificación (grado al menos 3).

Sabemos que en cualquier árbol el número de aristas es uno menos que el número de vértices, y que la suma de los grados de todos los vértices es 2 veces el número de aristas. Entonces,

$$t + 2s + 3r \leq \sum_{v \in V} gr(v) = 2|E| = 2(|V| - 1)$$

Pero $|V| = t + s + r$, entonces tenemos

$$t + 3r \leq 2(|V| - s - 1) = 2(t + r - 1)$$

Despejando, se sigue que $r \leq t - 2$ □

Corolario 3.1. *Sea G una gráfica conexa y S un conjunto de vértices terminales que contenga a lo más λ elementos. Podemos encontrar un árbol de Steiner analizando solo los subconjuntos de hasta $\lambda - 2$ vértices no-terminales de G.*

Demostración. Por el lema anterior, el conjunto de vértices de ramificación de cualquier árbol de Steiner para G y S tiene a lo más $\lambda - 2$ elementos.

Si nos fijamos en cada subconjunto de vértices no-terminales con $\lambda - 2$ elementos o menos, alguno de ellos debe corresponder al conjunto de vértices de

ramificación de un árbol de Steiner. Si juntamos cada uno de estos conjuntos con S y usamos el algoritmo codicioso, el teorema 3.2 nos dice que de los árboles que obtengamos, aquellos cuyo peso sea menor deben ser árboles de Steiner para G y S. □

3.4.2. Encontrando el Árbol de Steiner

Con estos resultados podemos plantear un conjunto de algoritmos que encuentran el árbol de Steiner en tiempo polinomial para conjuntos restringidos de vértices terminales. Si fijamos el valor λ, el siguiente algoritmo encuentra el árbol de Steiner de cualquier gráfica $G = (V, E)$ conexa y cualquier conjunto de vértices terminales S con λ elementos o menos. En el capítulo 4 demostraremos que la complejidad del algoritmo es polinomial, y depende del parámetro λ.

1. **Preprocesamiento.** Encontrar la distancia y una trayectoria de longitud mínima entre cada pareja de elementos de V.

2. **Fase de Búsqueda.** Para cada subconjunto R de vértices de $V - S$ tal que $|R| \leq \lambda - 2$, se hace lo siguiente:

   a) Calcular un árbol generador de peso mínimo para la gráfica $GP_G(S \cup R)$. Dependiendo del tipo de algoritmo que usemos para esto, es muy probable que tengamos que construir la gráfica de proximidad primero.

   b) Se registra el peso del árbol y se compara con el mínimo de los pesos registrados anteriormente. Si es menor, se establece como el nuevo mínimo, y si no se descarta.

   En esta fase es bueno mantener una referencia a un árbol cuyo peso sea igual al mínimo encontrado. Esta referencia se actualizará constantemente.

3. **Terminación.** Sea T el árbol de menor peso que encontramos en la fase anterior. Usando T, debemos construir un árbol en G que contenga a todos los elementos de S.

   Similarmente a como describimos antes, lo que hay que hacer es tomar todas las trayectorias de G inducidas por las aristas de T. Como consecuencia del teorema 3.2, la gráfica que resulta no puede contener ciclos (si tuviera alguno, al romperlo obtendríamos una gráfica más ligera, lo cual contradiría el teorema).

3.5. Algunas Observaciones

Los resultados que presentamos en este capítulo son bastante teóricos, pues nos hemos enfocado en demostrar que los algoritmos funcionan. Todos ellos se pueden describir con mucho más detalle, haciendo observaciones sobre las decisiones que hay que tomar en la manera de implementarlos, de forma que su

funcionamiento se optimice lo más posible. Este será el tema que trataremos en el siguiente capítulo.

Los algoritmos que presentamos al final, que construyen el Árbol de Steiner de conjuntos limitados de vértices terminales, son en realidad poco prácticos debido a que su complejidad aumenta considerablemente conforme tomamos conjuntos más y más grandes. Lo que nos interesa de estos resultados es que ponen en perspectiva las ideas que hemos estado manejando para resolver el problema.

Concretamente, hemos visto que la técnica codiciosa usada para construir el árbol no es suficiente, pero sí nos proporciona una manera sencilla de aproximar la solución. Cuando tenemos información extra sobre la forma del árbol (i.e. conocemos cuales son sus vértices de ramificación) entonces sí, la técnica codiciosa funciona bien.

De aquí en adelante daremos preferencia a las aproximaciones al árbol de Steiner, por ser más prácticas en la solución de problemas reales. De todas formas vale la pena tener en cuenta los otros resultados, pues en ellos radica una comprensión más profunda del problema.

Por último, obsérvese que todos los resultados de este capítulo son válidos aún cuando los pesos de las aristas de la gráfica son diferentes entre sí. Basta con que la distancia entre cualquier pareja de vértices esté bien definida. Los detalles de las demostraciones son prácticamente iguales, lo único que se complica un poco es la notación.

Capítulo 4

Implementación

Los algoritmos que hemos construído con relación al problema del Árbol de Steiner se pueden implementar de muchas maneras diferentes. De hecho, la forma en que los presentamos nos permite modularizarlos en tareas más específicas. Cada una de estas tareas se puede resolver de diversas maneras, cada una con sus propias ventajas y desventajas.

Este capítulo tiene dos objetivos. El primero es mostrar técnicas principales para resolver cada una de las partes que conforman al algoritmo. Usando estas técnicas podemos realizar el análisis de complejidad de los algoritmos del capítulo anterior en términos generales. El segundo objetivo es el de presentar, brevemente, las distintas opciones que hay en cada paso de la implementación, así como referencias bibliográficas que puedan proporcionar más información. Se puede pensar en este capítulo como un listado de piezas con las cuales podemos ensamblar los algoritmos que nos interesan; dependiendo del tipo de pieza que se use, el resultado funcionará mejor en unos casos que en otros.

En las dos primeras secciones estudiaremos cómo encontrar las distancias entre vértices en una gráfica. Nuestro algoritmo principal en este caso es BFS, pues es simple y muy eficiente, pero mencionamos también otras opciones. La primera sección trata el problema de encontrar las distancias entre un vértice y todos los demás. En cambio, la segunda sección estudia el problema de encontrar las distancias entre toda pareja de vértices de la gráfica.

En la tercera sección enfocamos nuestro estudio sobre el problema de ordenación. En particular, nos interesa mostrar la cota teórica de $O(n \log n)$ para ordenar un conjunto de n elementos, para lo cual usamos el algoritmo Merge-Sort como caso de estudio.

El otro problema que tenemos que estudiar es el de los Árboles Generadores de Peso Mínimo. Para esto, en la cuarta sección analizamos el algoritmo de Kruskal a profundidad, así como algunas otras propiedades de este tipo de árboles.

En la quinta sección presentamos el análisis de los algoritmos del capítulo anterior, el cual se basa en los métodos descritos en las secciones anteriores.

Concluimos con algunas observaciones de consideración sobre la implementación.

Recordemos que nuestro objetivo principal es el de estudiar el problema del Árbol de Steiner en gráficas donde el peso de las aristas no nos preocupa, sino más bien la cantidad de ellas que necesitamos. Esto es importante, pues determina cuáles algoritmos describimos más detalladamente. Finalmente, los libros [4, 14] son excelentes referencias para el material que cubrimos en este capítulo.

Sobre la notación: haremos referencia constantemente al número de aristas y vértices de una gráfica, para lo cual utilizaremos las letras q y p, respectivamente.

4.1. Búsqueda a lo Ancho (BFS)

Existen muchas maneras de recorrer una gráfica, y algunos de estos recorridos poseen propiedades muy interesantes. El algoritmo de búsqueda a lo ancho[1] describe uno de los métodos más importantes que hay.

El objetivo de un recorrido es visitar todos los vértices que posee la gráfica. Para esto, por lo general se escoge un vértice inicial desde el cual se trata de *alcanzar* a todos los demás. Esto es, se trazan caminos desde el vértice inicial hacia el resto. Cuando la gráfica es conexa, siempre se puede alcanzar a todos los vértices, y cuando no, se define un nuevo vértice inicial entre los que no se pudieron alcanzar, y el recorrido continúa desde ahí.

La idea del algoritmo BFS es recorrer los vértices por niveles. Los niveles forman una partición de los vértices y tienen la propiedad de que aquellos que pertenecen al nivel i se encuentran a distancia i del vértice inicial. El recorrido se hace de forma que mientras no se hayan visitado todos los elementos de un nivel, no se visitan los del siguiente. El algoritmo toma su nombre precisamente de aquí, pues le da mayor prioridad a la amplitud que a la profundidad.

Cuando visitamos un vértice v registramos la arista (u, v) que nos llevó a el, y decimos que u es *padre* de v (o v es *hijo* de u) en el recorrido. Todas estas aristas juntas forman una gráfica, que llamaremos D, de trayectorias mínimas entre el vértice inicial y todos los demás. Para ver por qué, fijémonos en un vértice cualquiera y recorramos la arista de D que nos lleva a su padre, y luego la que nos lleva al padre de este, y así sucesivamente hasta el vértice inicial. Cada arista de este camino salta del nivel del hijo al del padre, que es anterior. Por la forma en que definimos los niveles, dicho camino solo puede ser una trayectoria mínima. Esto también implica que cuando la gráfica original es conexa, D debe ser conexa, pues cada vértice se puede conectar con el vértice inicial. Además, como añadimos una sola arista por cada vértice, excepto el inicial, la gráfica que construimos no puede contener ciclos. Es decir, D debe ser un árbol.

A continuación mostramos una forma en que se puede implementar el algoritmo. La información de entrada que necesitamos es una gráfica $G = (V, E)$ y un vértice inicial $v_0 \in V$. Los niveles se van construyendo conforme progre-

[1] En inglés, Breadth First Search

sa el recorrido y la gráfica D se construye arista por arista. Los pasos son los siguientes:

1. Marcamos a v_0 como visitado y lo añadimos a una nueva lista. Esta lista representa el nivel 0 y es la que procesaremos primero en el siguiente paso. Al resto de los vértices los marcamos como no-visitados. Creamos otra lista, inicialmente vacía, donde colocaremos a los vértices del nivel 1. Inicializamos a D como una gráfica vacía.

2. Sean L la lista a la cual toca ser procesada y L' la lista que representa al siguiente nivel. Por cada arista $(u, v) \in E$ tal que $u \in L$, si v no ha sido visitado todavía lo marcamos como visitado, lo agregamos a L', y añadimos la arista (u, v) a D.

   En otras palabras, en este paso creamos la lista L' de todos los vértices de G que aun no hayan sido visitados y que se encuentran a distancia 1 de algún vértice de la lista L.

3. Si la lista L' construida en el paso anterior es vacía, el algoritmo termina. Si no, definimos a L' como la nueva lista a procesar, inicializamos una nueva lista vacía donde poner los vértices del siguiente nivel, y repetimos desde el paso 2.

La figura 4.1 muestra la ejecución de este algoritmo sobre una gráfica de ejemplo.

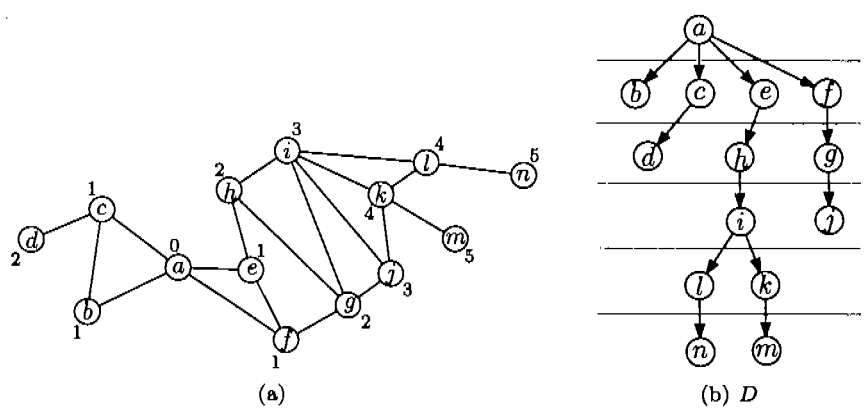


Figura 4.1: Una corrida del algoritmo BFS sobre una gráfica de ejemplo. El vértice de partida es a, y los números representan las distancias calculadas. D contiene las trayectorias mínimas desde a al resto de la gráfica.

El siguiente resultado nos permite concluir que el algoritmo anterior cumple con las condiciones de BFS que mencionamos antes.

Lema 4.1. *Después de la i-ésima iteración del paso 2 del algoritmo, todos los vértices a distancia i de v_0 han sido visitados.*

Demostración. Obsérvese que el resultado es trivial cuando $i = 0$, pues el único vértice a distancia 0 de v_0 es el mismo.

Si $i > 0$, sea $v \in V$ tal que $\delta(v_0, v) = i$. Si nos fijamos en un camino $\{v_0, v_1, \ldots, v_i = v\}$ de longitud mínima entre v_0 y v, entonces el sub-camino $\{v_0, v_1, \ldots, v_{i-1}\}$ debe ser mínimo entre v_0 y v_{i-1}, es decir $\delta(v_0, v_{i-1}) = i - 1$. Usando un argumento inductivo, la arista (v_{i-1}, v) debe ser analizada en la i-ésima iteración, lo cual nos asegura que v sea visitado. □

Una consecuencia de este lema es que el algoritmo construye los niveles correctamente. La lista que construye en la i-ésima iteración no contiene ningún vértice a distancia menor que i, pues estos ya fueron visitados en iteraciones anteriores. Tampoco contiene vértices a distancia mayor que i, pues las aristas que considera en cada iteración están enraizadas en los vértices de la lista de la iteración anterior, por lo que en i iteraciones no puede llegar a vértices a distancia mayor a i.

Corriendo BFS sobre una gráfica conexa obtenemos información muy útil: un árbol generador de la gráfica, trayectorias de peso mínimo entre el vértice inicial y todos los demás, y la distancia del vértice inicial a todos los demás. Todo esto en una forma compacta y muy manejable, el árbol D.

Si utilizamos una representación de la gráfica con listas de adyacencia (donde cada vértice mantiene una lista de vértices a los que es adyacente), en el paso 2 del algoritmo solo necesitaremos considerar a los vértices adyacentes a aquellos que estén en la lista L. Esto hace que cada arista sea considerada exactamente 2 veces: una por cada uno de sus extremos. El resto de las operaciones toma tiempo constante, por lo que el tiempo total de ejecución del algoritmo es $O(q)$.

4.1.1. El Algoritmo de Dijkstra

Se puede generalizar el recorrido BFS para que considere pesos positivos en las aristas de la gráfica. El resultado es el algoritmo de Dijkstra para encontrar las distancias (y trayectorias mínimas) desde un vértice inicial al resto.

Este algoritmo parte del vértice inicial y va haciendo crecer el conjunto de vértices visitados uno a la vez. En cada paso escoge de entre la "frontera" del conjunto[2] al vértice que sea más cercano al inicial, lo marca como visitado, y recalcula la frontera.

Usando colas de prioridades, es bastante simple dar una implementación del algoritmo que trabaje en tiempo $O(p^2)$. Mediante estructuras de datos más complejas se puede reducir hasta a $O(p \log p + q)$, lo cual es una mejora si la gráfica no tiene demasiadas aristas. Por lo tanto, este algoritmo es una buena opción cuando nos interesa resolver el caso en que las aristas de la gráfica tengan un peso positivo asignado.

Sin embargo, cuando hay aristas de peso negativo en la gráfica el algoritmo de Dijkstra puede fallar. La figura 4.2 ilustra este punto. El problema es que el algoritmo descarta las trayectorias que cerca del vértice inicial parecen muy pesadas, pero que después contienen aristas de peso negativo que les bajan

[2] I.e., los vértices no visitados que sean adyacentes a algún visitado

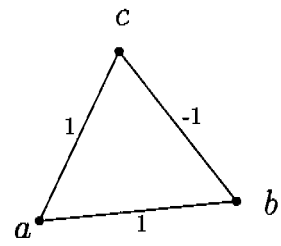


Figura 4.2: En esta gráfica el algoritmo de Dijkstra no puede encontrar la distancia de a al resto de los vértices. Ambas distancias son 0, pero una ejecución de Dijkstra haría lo siguiente: visitaría a b y marcaría la distancia a este como 1; luego tomaría la arista (b, c) y marcaría la distancia a c como 0; en ese momento ya no hay vértices sin visitar, por lo tanto el algoritmo termina, pero la distancia calculada al vértice b está mal. Si el algoritmo visitara primero al vértice c, el argumento es simétrico, y la distancia a c quedaría registrada como 1.

el peso total. Para cuando el algoritmo esta en posición de darse cuenta de esto, los vértices afectados ya fueron marcados como visitados, y por lo tanto son ignorados. En este caso la estrategia codiciosa no es la correcta. Más bien tendríamos que usar algún otro algoritmo, como el de Bellman-Ford.

4.1.2. El Algoritmo Bellman-Ford

Cuando una gráfica tiene algún ciclo de peso negativo se vuelve imposible definir los caminos de peso mínimo entre algunos vértices, pues cualquier camino que toque el ciclo se puede quedar dándole vueltas tantas veces como quiera, y cada vuelta le baja más el peso. En este caso ningún algoritmo nos puede ayudar. Si en cambio no hay ciclos de este tipo, el algoritmo Bellman-Ford sirve para encontrar las trayectorias de peso mínimo de una gráfica, aun cuando el peso de algunas de sus aristas sea negativo.

La idea principal que lo hace funcionar es la misma que en otros algoritmos de este tipo: cualquier subtrayectoria de una trayectoria de peso mínimo debe ser a su vez de peso mínimo. El algoritmo utiliza una técnica a la que se le llama "de relajación", en la cual cada vértice registra una distancia *tentativa* a la que se encuentra del vértice inicial. Conforme se van procesando las aristas, estas distancias tentativas se van haciendo más y más precisas (se ajustan, por asi decir), y la gráfica poco a poco va convergiendo a una en la cual cada vértice conoce su distancia real al vértice inicial.

El método de relajar aristas se muestra en la figura 4.3. Originalmente todos los vértices registran la distancia tentativa ∞ (excepto el vértice inicial, cuya distancia a si mismo es 0), indicando con esto que aun no encontramos un camino que nos lleve a ellos. Si pasamos por todas las aristas de la gráfica, relajando cada una en turno, las distancias tentativas registradas por cada vértice se ajustan un poco. Al repetir esto, las distancias se ajustan un poco más. De hecho, esta técnica tiene la propiedad de que después de hacer k pasadas sucesivas sobre

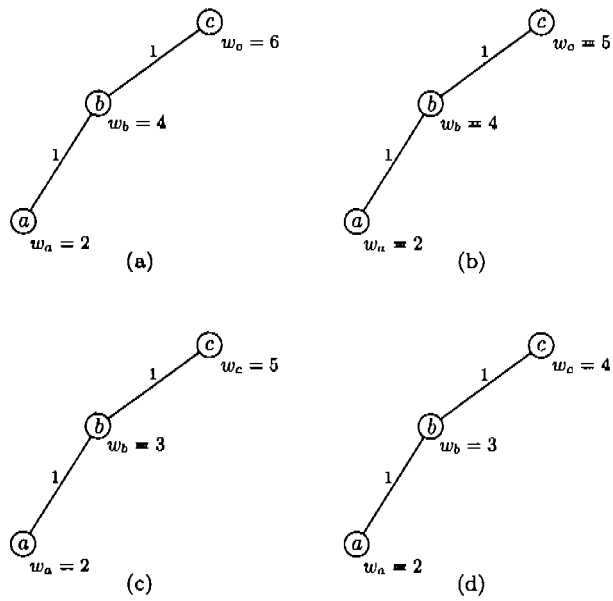


Figura 4.3: El método de relajación de aristas. Representamos por w_i la distancia tentativa registrada para el vértice i. En (a) se muestra la gráfica en su estado inicial.
En (b) se relaja la arista (b, c) que tiene peso 1, y como $w_c > w_b + 1$, el peso de w_c se actualiza al valor 5, que es la suma de w_b y el peso de la arista. En este momento se ha encontrado una trayectoria más corta que llega a c.
(c) Se relaja la arista (a, b), por lo que el nuevo valor de w_b es 3.
(d) Por último se relaja nuevamente (b, c), lo cual ocasiona un nuevo cambio de valor para w_c.

todas las aristas, se encuentran todos aquellos caminos de peso mínimo que partan del vértice inicial y que tengan k aristas o menos.

Cuando no hay ciclos de peso negativo en una gráfica, los caminos de peso mínimo no pueden contener ciclos, pues si así fuera, rompiendo el ciclo se llegaría a un camino más ligero. Esto quiere decir que para toda pareja de vértices, cualquier camino de peso mínimo que los una debe tener a lo más $p - 1$ aristas. El algoritmo Bellman-Ford hace $p - 1$ pasadas sobre las aristas de la gráfica, encontrando así las distancias y caminos correspondientes desde el vértice inicial al resto. Claramente, el algoritmo tiene una complejidad de $O(pq)$. La complejidad extra es el precio a pagar por tener aristas de peso negativo.

4.2. Camino Mínimo entre todo Par de Vértices

Los algoritmos de la sección anterior nos sirven para encontrar los caminos mínimos entre algún vértice particular y el resto de vértices de la gráfica. Si nos interesa tener esta información *para toda* pareja de vértices, hay varias maneras de lograrlo.

La primera es la más directa: para cada vértice, aplicar BFS, Dijkstra o Bellman-Ford, según si la gráfica tiene o no pesos en sus aristas, o si tiene pesos negativos. Esto nos da un conjunto de árboles que juntos contienen la información de los caminos mínimos que requerimos. La complejidad de este método depende del algoritmo que usemos: $O(pq)$ para BFS, $O(p^3)$ para Dijkstra,[3] y $O(p^2q)$ en el caso del algoritmo Bellman-Ford.

Otra opción es utilizar alguno de los algoritmos creados especialmente para resolver este problema. Dos de los principales son el algoritmo Floyd-Warshall, y el algoritmo de Johnson. Las ideas que utilizan son mucho más refinadas que el método del párrafo anterior.

Floyd-Warshall utiliza la riqueza de estructura que poseen los caminos mínimos, es decir, que si w es un vértice intermedio en un camino mínimo entre dos vértices u y v, entonces los subcaminos $u \rightsquigarrow w$ y $w \rightsquigarrow v$ son también mínimos. El truco entonces es seleccionar w inteligentemente de forma que no se hagan cálculos innecesarios. Aunque parece complicado, es bastante sencillo de implementar, y el resultado es un algoritmo de tiempo $O(p^3)$ que funciona también con aristas de pesos negativos (siempre y cuando no haya ciclos negativos).

El algoritmo de Johnson usa una técnica diferente. Primero calcula una nueva función de pesos para las aristas, de forma que todos los pesos se hagan positivos, pero preservando los caminos mínimos de la gráfica (i.e. un camino mínimo en la gráfica original induce un camino mínimo en la gráfica modificada y viceversa). A continuación corre el algoritmo de Dijkstra desde cada vértice y encuentra así la solución. Utilizando la implementación más avanzada del algoritmo de Dijkstra, el tiempo total para este método es $O(p^2 \log p + pq)$, que es mejor que Floyd-Warshall para gráficas con pocas aristas.

Aunque en general es más rápido utilizar los algoritmos especializados, la técnica simple que mencionamos al principio tiene una gran ventaja. Supongamos que tenemos un subconjunto R de vértices de la gráfica, con $|R| = r$, y que solo nos interesa conocer los caminos mínimos entre parejas de vértices que tengan la forma $\{(u, v) | u \in R\}$. En este caso, podemos correr BFS (o Dijkstra, o Bellman-Ford) r veces, usando a cada uno de los elementos de R como origen. El algoritmo que resulta tiene una complejidad de $O(rq)$ (o bien $O(p^2r)$ si usamos Dijkstra, $O(pqr)$ si usamos Bellman-Ford), que es muy buena cuando r es asintóticamente menor que p, por ejemplo, si $r = O(\sqrt{p})$.

[3] Aquí suponemos la implementación más sencilla del algoritmo, que corre en tiempo $O(p^2)$, pues la que utiliza estructuras de datos más complejas suele ser poco práctica.

4.3. Merge-Sort

El problema de ordenar los elementos de una lista ha sido ampliamente estudiado, y los algoritmos que lo resuelven son muy variados. Uno de los más conocidos es Merge-Sort. La idea fundamental que lo hace funcionar es la siguiente: supongamos que tenemos dos listas ordenadas de números[4] y queremos obtener una sola lista ordenada. El número más pequeño de todos es el primer elemento de alguna de las dos listas. Si quitamos este elemento, el siguiente más pequeño vuelve a estar como primer elemento de alguna de las dos listas (una de ellas fue reducida). Repitiendo hasta que alguna de las dos listas se termine podemos obtener todos los números en orden, y en cada paso solo tenemos que realizar una comparación. A este proceso se le conoce como *fusionar* listas y es de donde el algoritmo toma su nombre. Si entre las dos listas se tienen n números, en el peor caso hay que hacer $n-1$ comparaciones, por lo tanto fusionar dos listas tiene una complejidad lineal respecto a su entrada. En la figura 4.4 se muestra como funciona este método.

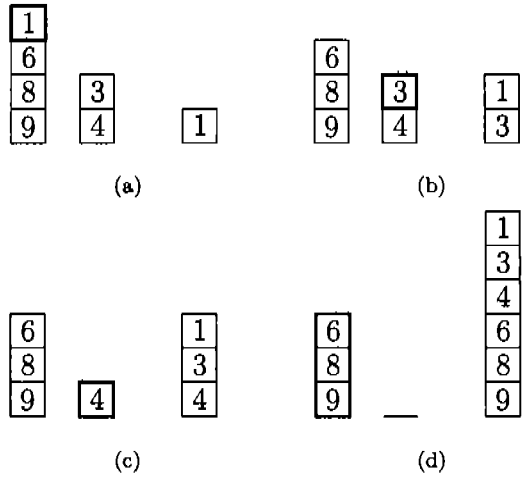


Figura 4.4: La tercera columna representa la fusión de las primeras dos. En cada paso se elige el elemento más pequeño de las listas, que debe estar al principio de alguna de ellas.

El algoritmo Merge-Sort toma como entrada una lista con n números y hace lo siguiente:

1. Separa los números en n listas, cada una conteniendo un solo elemento. Obsérvese que cualquier lista con un elemento está trivialmente ordenada.

2. Organiza el conjunto de listas por parejas y fusiona individualmente cada pareja. El resultado es un nuevo conjunto de listas ordenadas, cada una

[4]Por simplicidad, para la presentación de este algoritmo utilizaremos números, aunque realmente funciona con cualquier conjunto de objetos que se puedan ordenar totalmente.

del doble de tamaño que las anteriores. El número de listas al terminar este paso se reduce a la mitad.

3. Si solo queda una lista, ya terminamos. Si no, se repite desde el paso 2.

La correctez del algoritmo se sigue inmediatamente del hecho de que en cada paso las listas están ordenadas, pues el proceso que las fusiona las ordena. Se puede observar un ejemplo de la ejecución del algoritmo en la figura 4.5.

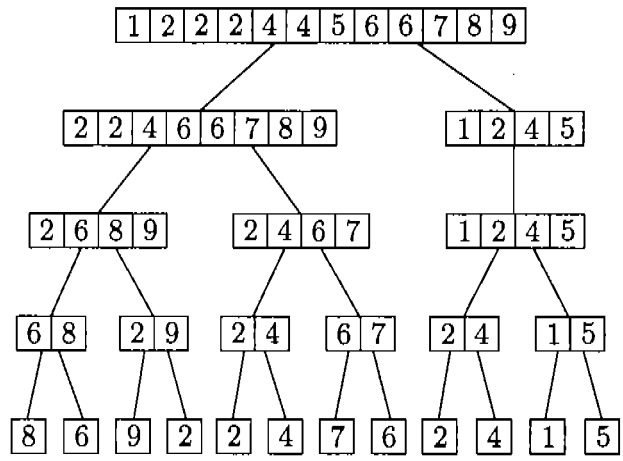


Figura 4.5: Un ejemplo de la ejecución de Merge-Sort. Comenzando desde abajo en la figura, se fusionan las listas de 2 en 2 hasta obtener una sola lista ordenada.

Para analizar su complejidad, supondremos que $2^{h-1} < n \leq 2^h$, de forma que $h - 1 < \log_2(n) \leq h$. Obsérvese que en cada iteración del paso 2, por cada comparación que se realiza, una de las listas de la iteración anterior pierde un elemento. Por lo tanto el número máximo de comparaciones en una sola iteración no pasa de n. Como mencionamos anteriormente, en cada iteración el número de listas se reduce a la mitad. Dado que originalmente hay n listas, al terminar la k-ésima iteración quedan $\lceil n/2^k \rceil$. Luego entonces después de h iteraciones

$$\frac{1}{2} < \frac{n}{2^h} \leq 1$$

i.e. $\lceil n/2^h \rceil = 1$ y el algoritmo termina. En conclusión, el tiempo de ejecución de Merge-Sort es $O(nh)$, o lo que es lo mismo, $O(n \log n)$.

Existen muchos otros algoritmos de ordenación. Algunos explotan propiedades particulares de la entrada; por ejemplo, si se puede probar que ningún elemento es más grande que algún número M dado, esta información se puede utilizar para ordenar los números más eficientemente. Para el caso general en que no sabemos la forma de la entrada y lo único que podemos hacer es comparar parejas de elementos, se ha probado que se necesitan al menos $O(n \log n)$ comparaciones en el peor caso, para *cualquier* algoritmo que resuelva el problema.

En este sentido Merge-Sort es óptimo. En la práctica, sin embargo, existen algoritmos que son más rápidos salvo por algunas entradas patológicas. Un ejemplo notable es Quicksort, cuyo peor caso es $O(n^2)$, pero que en el caso promedio corre más rápido que muchos otros algoritmos con mejores cotas teóricas.

El punto importante de esta discusión es que el problema de ordenación se puede resolver en $O(n \log n)$ en el peor caso, que es la cota que usaremos en el análisis posterior. Para aplicaciones prácticas en las cuales sea tolerable tener corridas más lentas de vez en cuando, Quicksort puede ser una buena opción.

4.4. Algoritmo de Kruskal

Una gran variedad de problemas de optimización en gráficas utilizan, directa o indirectamente, los algoritmos desarrollados para encontrar el Árbol Generador de Peso Mínimo. Se trata de obtener una subgráfica que les permita a todos los vértices seguir conectados unos con otros, probablemente eliminando algunas aristas, de forma que se reduzca el peso total lo más posible.

El algoritmo de Kruskal resuelve el problema construyendo el árbol arista por arista. En cada paso, elige la arista más ligera que al introducirla no cree ciclos entre las aristas añadidas anteriormente. Esta estrategia es codiciosa, pues la elección minimiza el peso de la gráfica lo más posible a cada paso. Aunque esta estrategia no siempre resuelve el problema correctamente, como ya hemos visto, en este caso si funciona. Lo que sigue es una descripción más detallada del algoritmo, que trabaja sobre una gráfica conexa:

1. Se coloca cada vértice de la gráfica en un conjunto, de forma que se tengan $|V|$ conjuntos disjuntos de un solo elemento. Durante su ejecución, el algoritmo mantendrá la siguiente invariante: cada conjunto representa una componente conexa de la solución parcial. De esta forma, las aristas que no hayan sido añadidas crearían un ciclo si y solo si tuvieran sus dos extremos en un mismo conjunto.

2. Se ordenan las aristas de la gráfica por peso, de menor a mayor.

3. Considerando cada arista en turno, empezando por la menor y yendo en orden hacia la mayor, por cada una se aplica el siguiente criterio: si los dos extremos de la arista están en el mismo conjunto, se descarta (i.e. el proceso continua con la siguiente arista sin hacer cambios a la solución parcial); si al contrario, los extremos están en conjuntos distintos, se añade la arista a la solución parcial y los conjuntos correspondientes son unidos en uno solo (esto refleja como la nueva arista junta las dos componentes conexas en una).

4. El algoritmo termina cuando queda un solo conjunto. Las aristas seleccionadas en el paso 3 conforman la solución.

En la figura 4.6 se aprecia la ejecución del algoritmo sobre una gráfica de ejemplo.

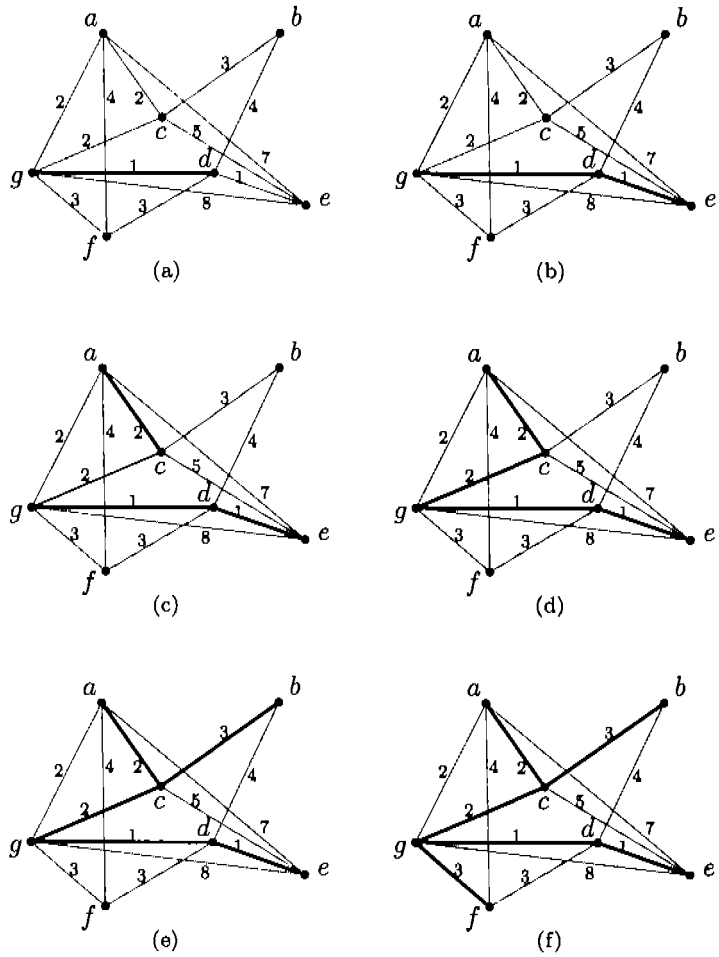


Figura 4.6: Ejecución del Algoritmo de Kruskal sobre una gráfica. Obsérvese que en (e), la arista (a, g) es la más ligera, pero al añadirla crearía un ciclo, por lo que se elige en cambio la arista (b, c).

Observación 4.1. Dado que la gráfica es conexa, la presencia de más de un conjunto en cualquier momento de la ejecución del algoritmo implica que hay alguna arista que no ha sido considerada. Se sigue que después de procesar todas las aristas siempre queda un solo conjunto.

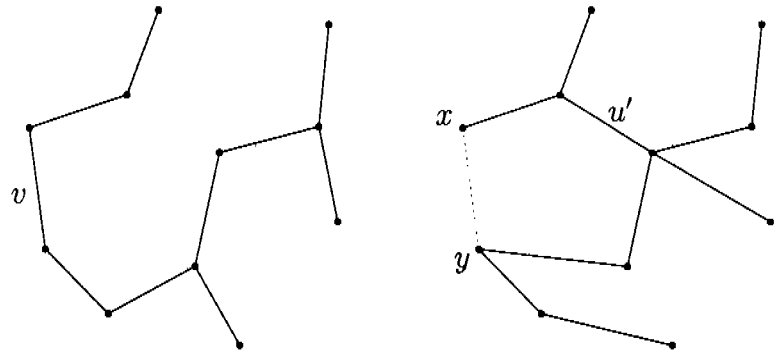
Esto significa que la gráfica que se obtiene es conexa, contiene a todos los vértices, y por construcción no contiene ciclos, por lo tanto es un árbol generador. El siguiente resultado muestra que se trata de un Árbol Generador de Peso Mínimo.

Lema 4.2. *Sea $G = (V, E, w)$ una gráfica con pesos en las aristas y T un Árbol Generador de Peso Mínimo de G. Si T_K es el árbol construído por el algoritmo de Kruskal, entonces $w(T) = w(T_K)$.*

Demostración. Mostraremos que el árbol T se puede llevar a T_K intercambiando aristas, de forma que el peso no cambie.

Si T y T_K tienen las mismas aristas, el resultado se sigue trivialmente. Si no, sea u una arista de peso mínimo que esté en T_K pero no en T. Si añadimos u a T, se debe generar un ciclo que contiene a u. También, en dicho ciclo debe haber al menos una arista v que esté en T pero no en T_K (si no, el ciclo estaría presente en T_K). Por lo tanto debe cumplirse que $w(v) \leq w(u)$, o de lo contrario sustituyendo a v por u en T obtendríamos un árbol generador más ligero, lo cual es imposible.

Similarmente, si añadimos la arista v a T_K, se genera un ciclo que contiene a v. Sea u' alguna arista del ciclo que no esté en T (debe haber al menos una). En T_K, la trayectoria *única* que va de un extremo de v al otro tiene que pasar por u' (véase la figura 4.7). Por lo tanto, en el momento en que el algoritmo de Kruskal considera a la arista u' en el paso 3, los extremos de v se deben encontrar en componentes conexas distintas. Esto implica que $w(u') \leq w(v)$, pues de lo contrario v habría sido considerada antes por el algoritmo, y habría sido añadida a T_K. Pero por la manera como elegimos a u, se tiene que $w(u) \leq w(u') \leq w(v)$.



(a) T (b) T_K

Figura 4.7: Al añadir la arista v a T_K se genera un ciclo, el cual contiene al menos una arista u' que no está en T. En $T_K \setminus \{u'\}$ los extremos de la arista v están en componentes conexas distintas.

En conclusión, $w(u) = w(v)$, por lo que si intercambiamos la arista v por u en T, obtenemos un nuevo árbol T' de forma que $w(T) = w(T')$. Además, el número de aristas en común con T_K es mayor en T' que en T. Repitiendo este proceso hasta que los dos árboles sean el mismo, llegamos a la conclusión de que $w(T) = w(T_K)$, i.e. T_K es de peso mínimo. □

La complejidad del algoritmo de Kruskal depende fuertemente de la manera en que representemos los conjuntos disjuntos. Hay tres operaciones a implemen-

tar. Primero, debe ser posible crear un nuevo conjunto; esta operación se puede llevar a cabo en $O(1)$. Dado un elemento, necesitamos saber a qué conjunto pertenece; esto lo podemos hacer en $O(\log p)$. Por último, debemos poder unir dos conjuntos en uno; hacerlo tiene complejidad $O(1)$. Más adelante explicaremos la manera en que se implementan estas operaciones y probaremos sus complejidades respectivas.

En total se construyen p conjuntos, y se utiliza la operación de unión $p-1$ veces (en cada caso reduciendo el número de conjuntos en 1). Por cada arista, necesitamos saber a qué conjuntos pertenecen sus dos extremos, por lo que en el peor caso necesitamos $2q$ consultas de este tipo. En total el tiempo necesario por las operaciones con conjuntos es de $O(p+q\log p)$ o lo que es lo mismo, $O(q\log p)$ (como la gráfica es conexa, $q \geq p-1$). Finalmente, ordenar las aristas se lleva $O(q\log q)$ usando Merge-Sort, pero $q < p^2$, luego entonces $\log q < 2\log p$. Al juntar todo nos queda que el tiempo requerido por el algoritmo para encontrar el Árbol Generador de Peso Mínimo es $O(q\log p)$.

4.4.1. Conjuntos Disjuntos

Para modelar a los conjuntos utilizaremos árboles. La cabeza del árbol servirá como representante del conjunto, por lo que para saber a qué conjunto pertenece un elemento basta recorrer su árbol hasta la raíz. Con esta representación, crear un nuevo conjunto con un solo elemento es trivial y el tiempo requerido es $O(1)$.

Para unir dos conjuntos, es suficiente con hacer que la raíz de alguno de ellos apunte a la raíz del otro. Esta operación también la podemos llevar a cabo en tiempo constante.

En el peor caso, averiguar a qué conjunto pertenece un elemento lleva tiempo proporcional a la altura del árbol. Si este está mal construído, puede ser lineal en el número de elementos (cuando el árbol es realmente una trayectoria). Sin embargo hay una técnica sencilla que nos permite mantener relativamente pequeña la altura del árbol. Por cada árbol llevamos un registro de su altura, el cual actualizaremos cuando hagamos operaciones de unión entre los conjuntos. A este dato le llamaremos *rango*. Asi, por ejemplo, un árbol con un solo elemento tiene rango 0. La idea es hacer que al unir conjuntos, el de altura más pequeña quede como subárbol del de altura más grande, así la altura cambia lo menos posible.

Cuando se unen dos árboles de rango distinto, el nuevo árbol conserva la raíz y el rango del mayor. Si los dos árboles tienen el mismo rango, elegimos arbitrariamente la raíz de alguno de ellos como la nueva raíz, e incrementamos su rango en una unidad.

Lema 4.3. *Usando la técnica descrita anteriormente, un árbol que tenga rango k debe contener al menos 2^k elementos.*

Demostración. Lo demostraremos por inducción. Cuando el rango es 0, el conjunto tiene un solo elemento, por lo que en este caso el lema se cumple. Ahora bien, supongamos que para $n-1$, con $n > 0$, el árbol tiene tiene al menos 2^{n-1}

elementos. Demostraremos que en este caso la proposición es cierta también para n.

Sea T un árbol de rango n. Tenemos dos casos, cuando T fue construído a partir de dos árboles de rangos distintos o cuando fue construído a partir de dos árboles de rangos iguales.

Analizaremos primero el segundo caso. Por la forma en que definimos la unión, los dos árboles que se juntaron para crear a T deben ser de rango $n-1$, y por la hipótesis de inducción cada uno de ellos debe contener al menos 2^{n-1} elementos. Al juntarlos se sigue fácilmente que T tiene al menos $2(2^{n-1}) = 2^n$ elementos.

Si en cambio T fue construído por dos árboles de rangos distintos, uno de ellos debe haber sido de rango n. En este caso, ese árbol también lo podemos separar en los dos que lo construyeron, y si son de rangos iguales, usamos el argumento del primer caso; si no, uno de ellos debe ser de rango n. Repitiendo este razonamiento, y dado que originalmente hay un número finito de conjuntos, eventualmente tenemos que llegar a un subárbol de T que fue construído por dos del mismo rango y el resultado se sigue por el primer caso. Los árboles de rango menor a $n-1$ no nos estorban, porque incrementan aún más el número de elementos de T.

En conclusión, en ambos casos T tiene al menos 2^n elementos. □

Corolario 4.1. *Dado un elemento, saber a qué conjunto pertenece lleva tiempo $O(\log n)$, donde n es el total de elementos.*

Demostración. En el peor caso hay un solo conjunto que contiene a todos los elementos. El máximo número de aristas que hay que recorrer para llegar a la raíz (y así poder decir a qué conjunto pertenece un elemento) está acotado por el rango del árbol, que por el resultado anterior es a lo más $\lceil \log_2(n) \rceil$. □

Existe una implementación un poco más refinada que utiliza una técnica conocida como *compresión de caminos*. Con esto, averiguar el conjunto al que pertenece un solo elemento puede tomar tiempo $O(\log n)$, pero las consultas sucesivas a la estructura de datos tardan solo tiempo $O(1)$ en muchos casos. El resultado es que si se hacen muchas operaciones sobre la estructura, el tiempo amortizado por operación es casi constante (y para todo uso práctico se puede considerar constante).

4.4.2. Árboles de Peso Mínimo en Gráficas Dinámicas

Hasta ahora hemos supuesto que la gráfica sobre la que calculamos el Árbol Generador de Peso Mínimo no cambia. Un problema relacionado es el de actualizar el árbol cuando se da alguna modificación en la gráfica (por ejemplo, se añaden o eliminan vértices y aristas).

Supongamos que se le agrega un conjunto de aristas nuevas a una gráfica (incluso, posiblemente, agregando también algunos vértices). Mostraremos a continuación que podemos recalcular el árbol usando solamente el árbol anterior

y las nuevas aristas. Para esto necesitamos hacer primero algunas observaciones sobre la naturaleza de los Árboles Generadores de Peso Mínimo.

Los árboles son gráficas que tienen la propiedad de minimalidad respecto a la conexidad; esto es, mantienen la conexidad entre los elementos de la gráfica con un número mínimo de aristas. Por eso, cuando se introduce alguna arista v que tenga sus dos extremos dentro del árbol, se genera un ciclo. Naturalmente, el ciclo se encuentra compuesto por la arista v y otras aristas del árbol, a las cuales les llamaremos *aristas intercambiables con* v. Es sencillo convencerse de que si se intercambia v por alguna de estas aristas se obtiene un nuevo árbol que conecta a los mismos vértices que el anterior. Si además el árbol original es de peso mínimo, la arista v debe ser al menos tan pesada como cualquiera de las aristas con las que es intercambiable, o de lo contrario se podría hacer un intercambio y obtener un árbol más ligero. Estas observaciones forman el núcleo de la demostración del siguiente resultado.

Lema 4.4. *Sea G' una gráfica conexa, T' un Árbol Generador de Peso Mínimo de G', y E un conjunto de aristas ajenas a G'.*

Si $G' \cup E$ es conexa y T es un Árbol Generador de Peso Mínimo de la gráfica inducida por $T' \cup E$, entonces también es un Árbol Generador de Peso Mínimo para $G' \cup E$.

Demostración. Sea $G = G' \cup E$ y T_G un Árbol Generador de Peso Mínimo de G. Como T' es generador de G', T debe contener a todos los vértices de $G' \cup E$, luego es también un árbol generador. Para demostrar que tiene peso mínimo, iremos intercambiando aristas entre T y T_G sin alterar el peso. Mediante estos cambios, eventualmente llevaremos T_G a T, con lo que habremos terminado.

Por conveniencia, si dos *árboles* tienen los mismos vértices definimos la distancia entre ellos como el número de aristas que los separan, i.e. el número de aristas que están presentes en uno pero no en el otro (como el número de aristas es igual en ambos árboles, la distancia queda bien definida sin importar cual árbol tomemos como referencia). Por ejemplo, los árboles de la figura 4.7 están a distancia 4.

Obsérvese que la distancia entre dos árboles es 0 si y solo si son el mismo árbol. Supongamos entonces que la distancia entre T_G y T es $d > 0$. Encontraremos un árbol generador de G con el mismo peso de T_G, pero a distancia $d-1$ de T.

De entre las aristas de T que son ajenas a T_G, sea u una de peso mínimo, y sea v una de las aristas intercambiables con u en T_G que no se encuentre en T (debe haber al menos una, o de lo contrario el ciclo inducido por u en T_G estaría presente en T). Como establecimos antes, debe cumplirse que $w(u) \geq w(v)$.[5] Ahora separamos el análisis en dos casos.

Primer Caso. Cuando $v \in T' \cup E$, un argumento simétrico al anterior muestra que debe existir alguna arista u' intercambiable con v en T, de tal forma que

[5] Usamos $w(a)$ para representar el peso de la arista a.

u' no esté en T_G. También, $w(v) \geq w(u')$, y por la forma en que elegimos a u, $w(u') \geq w(u)$.

Segundo Caso. Cuando v no está en $T' \cup E$ no podemos simplemente añadirla a T y usar el mismo argumento de antes, pues no pertenece a la gráfica de la cual T es Árbol Generador de Peso Mínimo.

Lo que haremos es añadirla a T' y fijarnos en el conjunto D de las aristas que sean intercambiables con v y ajenas a T. Si este conjunto es vacío, entonces el ciclo generado por v en T' se debe formar también al añadir v a T; i.e., debe existir alguna arista u' de T intercambiable con v, de forma que $w(v) \geq w(u') \geq w(u)$ como antes.

Si en cambio existe alguna arista $x \in D$, podemos añadir x a T y formar un nuevo ciclo. Tenemos entonces dos ciclos: uno formado al introducir v en T' (del cual x es miembro), y otro al introducir x a T. En el primero v es una arista de peso máximo, y en el segundo x lo es, por lo que por transitividad v es de peso máximo entre todas las aristas de ambos ciclos. Como x pertenece a los dos ciclos (véase la figura 4.8), al unirlos y eliminar x obtenemos un nuevo ciclo en G que contiene a v.[6] Las aristas de dicho ciclo que sean intercambiables con v y que no estén en T deben ser elementos de $D \setminus \{x\}$. Podemos repetir el mismo proceso sucesivamente con el nuevo ciclo hasta que agotemos las aristas de D.

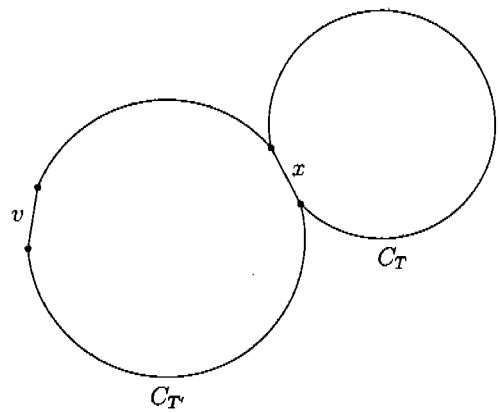


Figura 4.8: Los ciclos generados en T' y T al añadirles v y x, respectivamente. $v \in T_G$, y x es una arista de T' opuesta a v y ajena a T_G que además no está en T. En $(C_{T'} \cup C_T) \setminus \{x\}$ existe un ciclo que contiene a v como arista de peso máximo.

El ciclo final que obtengamos estará completamente contenido en T excepto por v, por lo que debe contener alguna arista u' que sea ajena a T_G, luego

[6] Estamos simplificando un poco el proceso. En realidad, es posible que al unir ambos ciclos y eliminar a x no obtengamos un ciclo, por ejemplo si los ciclos se intersectan en algún otro elemento. Sin embargo, no es muy complicado ver que de esta unión se puede extraer un ciclo que contiene a v, y este es el ciclo al que nos referimos en el texto.

$w(u') \geq w(u)$. Dado que v es de peso máximo en cada iteración del proceso, $w(v) \geq w(u') \geq w(u)$.

En conclusión, en ambos casos $w(v) \geq w(u)$, luego entonces $w(v) = w(u)$, por lo que si intercambiamos estas aristas en T_G obtenemos un nuevo árbol generador de G que pesa lo mismo, y cuya distancia a T es $d - 1$. Si seguimos intercambiando aristas de esta manera, eventualmente llegaremos a T sin modificar el peso, por lo tanto T es de peso mínimo para $G' \cup E$. □

El resultado anterior nos servirá para ahorrarnos cálculos en el caso de los algoritmos incrementales del capítulo 3.

4.4.3. Otras Opciones

Existen muchas maneras de calcular el Árbol Generador de Peso Mínimo de una gráfica. El algoritmo de Kruskal es una de las más sencillas y más directas, y para nuestros propósitos es bastante útil. Sin embargo, hay que tener en cuenta que se puede utilizar otro algoritmo cuando sea conveniente o necesario, por ejemplo, si la gráfica tiene alguna particularidad que se pueda explotar.

Para una discusión más avanzada, [2] es una buena referencia.

4.5. Árbol de Steiner

Lo que hemos discutido en las secciones anteriores nos servirá como herramienta para el análisis y discusión que hagamos en esta sección. Hablaremos primero sobre el algoritmo codicioso, su implementación y análisis de complejidad. Después analizaremos más brevemente los otros algoritmos de aproximación y el algoritmo que encuentra la solución exacta.

En lo que sigue, supóngase que estamos trabajando sobre una gráfica con p vértices, q aristas y r vértices terminales seleccionados.

4.5.1. 2-Aproximación

Teorema 4.1. *Para una gráfica conexa, sin pesos en las aristas, podemos construir la 2-aproximación al Árbol de Steiner en tiempo* $O(rq + r^2 \log r)$.

Demostración. El algoritmo codicioso que encuentra la 2-aproximación tiene que realizar las siguientes tareas:

1. Construir la gráfica de proximidad del conjunto de vértices terminales. Para esto necesita primero encontrar las distancias en G entre dichos vértices. Si se usa BFS desde cada vértice terminal, en total lleva $O(rq)$ obtener esta información y de paso se calculan también las trayectorias mínimas en G que unen a cada pareja de vértices.

   Teniendo las distancias, construir la gráfica de proximidad es equivalente a construir una gráfica completa usando a los vértices terminales donde cada arista (a, b) pesa $\delta_G(a, b)$. Esto se puede hacer en $O(r^2)$.

2. Encontrarle un Árbol Generador de Peso Mínimo a la gráfica de proximidad. Este paso toma un tiempo de $O(r^2 \log r)$ usando el algoritmo de Kruskal, pues la gráfica de proximidad tiene $\frac{r(r-1)}{2}$ aristas.

3. Construir la 2-aproximación en G a partir del árbol. Por cada arista del árbol, se debe listar una trayectoria de peso mínimo en G que una a sus extremos. Pero esto ya fue calculado en el primer paso, por lo que el costo total solo es el costo de listar las aristas de las trayectorias, lo cual lleva $O(q)$ en el peor caso.

Obsérvese que no solo afecta a la complejidad el tamaño de la gráfica, sino también el número de vértices terminales que se estén considerando. El tiempo total requerido es $O(rq + r^2 + r^2 \log r + q) = O(rq + r^2 \log r)$. Esto prueba el teorema. □

Si queremos correr varias veces el algoritmo sobre la misma gráfica, es posible que nos convenga *preprocesarla* para encontrar las distancias y trayectorias mínimas entre cada pareja de vértices (usando BFS desde cada vértice, el preprocesamiento se puede hacer en $O(pq)$). Así nos evitamos recalcular esta información cada vez que queramos construir una 2-aproximación. Si descartamos el tiempo necesario para el preprocesamiento, la complejidad del algoritmo se reduce a $O(r^2 \log r + q)$.

Si lo que nos interesa es medir la distancia respecto al peso de las aristas, o si la gráfica tiene alguna particularidad que se pueda explotar, los algoritmos que usamos en cada caso se pueden sustituir por otros que hagan una tarea similar. El tiempo total se verá afectado según estos cambios, pero el análisis debe ser muy simple usando un desglosado similar al que hicimos aquí.

4.5.2. Heurística del Conjunto Fundamental

Ahora vamos a analizar el algoritmo que busca mejorar la aproximación tratando de encontrar el conjunto fundamental del Árbol de Steiner elemento a elemento. Para buscar *1 solo vértice*, el algoritmo debe realizar las siguientes tareas:

1. Preprocesar la gráfica para obtener la información de distancias y trayectorias mínimas entre cada par de vértices, así como encontrar la 2-aproximación usando el algoritmo codicioso. Las técnicas son las mismas que las descritas para encontrar la 2-aproximación, por lo que este paso lleva $O(pq + r^2 \log r)$. En este caso, sin embargo, lo que más nos conviene es quedarnos con el Árbol Generador de Peso Mínimo de la gráfica de proximidad en vez de reconstruir la aproximación en G. Esto es porque lo vamos a utilizar para simplificar los cálculos en el siguiente paso.

2. Considerando cada vértice en turno, añadirlo al conjunto de vértices terminales y encontrar un Árbol Generador de Peso Mínimo para la gráfica de

proximidad de este nuevo conjunto. Seleccionar algún vértice cuyo árbol sea mínimo en peso respecto a los demás.

Conviene pensar en la gráfica de proximidad dinámicamente: le añadimos y quitamos vértices y aristas, y cada vez que hacemos esto debemos actualizar el Árbol Generador de Peso Mínimo. Es decir, por cada vértice debemos extender la gráfica (incluyendo al vértice y sus aristas incidentes) y recalcular el árbol. Según el lema 4.4, una manera de hacerlo es tomar las aristas del árbol original que calculamos en el primer paso y añadirles las aristas incidentes al vértice nuevo. De estas últimas hay r, y el árbol original tiene $r-1$, por lo que en total tenemos que considerar $2r-1$ aristas. De acuerdo al lema, un Árbol Generador de Peso Mínimo construido a partir de estas aristas lo es también de la gráfica de proximidad extendida. Por lo tanto usando el algoritmo de Kruskal se puede realizar este paso en tiempo $O(r \log r)$ por cada vértice (dado que ya tenemos calculadas las distancias entre todo par de vértices, listar las aristas incidentes al vértice nuevo lleva $O(r)$). Procesar todos los vértices de la gráfica de esta manera lleva $O(pr \log r)$, y conforme vamos procesando cada vértice podemos llevar un registro del que genera el árbol más ligero hasta el momento.

En total este proceso toma $O(pq + r^2 \log r + pr \log r) = O(pq + pr \log r)$, y al final tendremos un candidato a vértice de ramificación del árbol de Steiner.

Podemos volver a aplicar la heurística usando el conjunto de vértices terminales extendido con el vértice anterior para obtener otro posible vértice de ramificación del árbol. Como establecimos anteriormente, en total el árbol no puede tener más de $r-2$ vértices de ramificación, por lo que este es el máximo número de iteraciones para este proceso. Las iteraciones sucesivas pueden utilizar la información calculada en las iteraciones anteriores; en particular podemos reciclar el trabajo del preprocesamiento de los vértices, y el Árbol Generador de Peso Mínimo de la gráfica de proximidad extendida. Por esto, a partir de la segunda iteración solo es necesario realizar el paso 2 descrito arriba para el conjunto de vértices terminales que corresponda.

De todas las iteraciones, la más costosa es la última, que en el peor caso tiene que trabajar con $r+(r-3)$ vértices terminales. Aún así, la complejidad de esta iteracion es $O(pr \log r)$, pues al calcular los árboles de peso mínimo solo es necesario considerar poco menos de $4r$ aristas en cada caso. Toda esta discusión prueba el siguiente resultado.

Teorema 4.2. *La heurística del conjunto fundamental encuentra una aproximación al árbol de Steiner en tiempo $O(pq + pr^2 \log r)$.*

4.5.3. Solución Exacta

Por último analizamos el algoritmo que encuentra la solución exacta al problema del Árbol de Steiner, siempre y cuando el número de elementos del conjunto de vértices terminales esté acotado por un valor λ. Como mencionamos antes, en este caso el número de vértices de ramificación del árbol es a lo más

$\lambda - 2$, por lo que con considerar conjuntos de hasta este número de elementos de entre los no-terminales, necesariamente encontraremos el árbol.

Sea S el conjunto de vértices terminales de la gráfica. El algoritmo tiene que realizar las siguientes tareas:

1. Preprocesar la gráfica para obtener la información de distancias y trayectorias mínimas entre cada pareja de vértices.

2. Por cada subconjunto S' de vértices no-terminales de entre cero y $\lambda - 2$ elementos, obtener el Árbol Generador de Peso Mínimo para la gráfica de proximidad del conjunto $S \cup S'$. En cada caso registrar el peso del árbol asociado al conjunto. Escoger cualquiera de los conjuntos que minimicen dicho peso.

El primer paso ya lo hemos discutido: usando BFS desde cada vértice podemos llevarlo a cabo en $O(pq)$. Para analizar la complejidad del segundo paso vamos a simplificar las cosas un poco. En vez de considerar subconjuntos de hasta $\lambda - 2$ vértices no-terminales, supondremos que el algoritmo trabaja sobre todos los subconjuntos de exactamente $\lambda - 2$ vértices, incluyendo terminales y no terminales. Estas dos situaciones son equivalentes desde el punto de vista teórico, pues a cualquier subconjunto de vértices no-terminales le podemos agregar vértices terminales hasta que tenga $\lambda - 2$ elementos. Por lo tanto todos los subconjuntos considerados en el paso 2 siguen siendo considerados al hacer este cambio de enfoque.

El análisis que haremos a continuación podría ser mucho más estricto, y así arrojar mayor información sobre la naturaleza y complejidad de este algoritmo. Sin embargo nuestro objetivo en este caso es otro. Lo que queremos es mostrar que con la restricción extra de que el número de vértices terminales esté acotado, si se pueden encontrar soluciones al problema del Árbol de Steiner en tiempo polinomial.

El número de subconjuntos de $\lambda - 2$ elementos de entre p totales está dado por

$$\begin{aligned} C^p_{\lambda-2} &= \frac{p!}{(\lambda-2)!(p-(\lambda-2))!} \\ &= \frac{(p)(p-1)(p-2)\cdots(p-(\lambda-3))}{(\lambda-2)!} \\ &\leq \frac{p^{\lambda-2}}{(\lambda-2)!} \\ &\leq p^{\lambda-2} \end{aligned}$$

Por lo tanto en el paso 2 se deben considerar $O(p^{\lambda-2})$ conjuntos. Como $|S| \leq \lambda$, en total hay menos de 2λ elementos en $S \cup S'$ en cada iteración, por lo que encontrar su gráfica de proximidad toma $O(\lambda^2)$ y el cálculo del Árbol Generador de Peso Mínimo lleva $O(\lambda^2 \log \lambda)$ usando Kruskal. Juntando todos estos resultados obtenemos el siguiente lema.

Lema 4.5. *Si $r \leq \lambda$, i.e. el número de vértices terminales no pasa de cierta constante, entonces siempre podemos encontrar el Árbol de Steiner en tiempo* $O(pq + p^{\lambda-2}\lambda^2 \log \lambda)$.

En la práctica el algoritmo no es muy útil salvo para valores pequeños de λ. En otro caso sería mejor buscar aproximaciones al Árbol de Steiner, posiblemente usando alguna heurística. Desde el punto de vista teórico es relevante al menos en un sentido: hay que notar como un pequeño cambio en el planteamiento del problema nos lleva de algo que quizás no sea posible de resolver, a algo cuya solución es relativamente simple.

4.6. Sobre la Implementación

Muchos de los problemas que discutimos en este capítulo han sido ampliamente tratados con anterioridad. Los resultados sobre ellos son numerosos y los algoritmos que los resuelven son muy variados. La elección de una técnica en particular para resolver un problema puede cambiar dependiendo de las características particulares que se puedan explotar de ese problema.

En este capítulo nos hemos enfocado en implementar el algoritmo codicioso con simpleza y eficiencia, de la forma más general posible, lo cual nos ha llevado a dar preferencia a ciertos algoritmos sobre otros. Hay que tener en cuenta que dependiendo de la gráfica sobre la que se trabaja, puede ser conveniente cambiar de herramientas. La naturaleza modular de nuestra 2-aproximación hace que este punto sea particularmente importante.

Capítulo 5

Movilidad

En el mundo de las comunicaciones cotidianas, la suposición de que la red sobre la que se quiere trabajar es estática es muy poco realista: las computadoras cuentan con tarjetas de red inalámbricas, lo cual significa que la topología de la red se define en el momento de su uso (a este tipo de redes se les llama *ad-hoc*); los teléfonos celulares son el ícono de la movilidad, por lo que asumir que se mantendrán fijos mientras se estén comunicando es un error. El tipo de algoritmos que se deben desarrollar para este tipo de redes es necesariamente distinto, y tiene que acoplarse a este dinamismo.

A lo largo de este capítulo analizaremos el problema del Árbol de Steiner en gráficas "móviles", o más precisamente, gráficas para las cuales ignoramos cual es la distribución de sus nodos y aristas. En un extremo de esta abstracción el problema se vuelve imposible de resolver, cuando los vértices se mueven más rápido de lo que podemos decidir cuál se conecta con cuál. Pero también este modelo es poco realista, pues en general los nodos se mueven varios órdenes de magnitud más despacio que las señales que transmiten. Por eso casi siempre es considerado aceptable que los algoritmos desarrollados para este tipo de problemas trabajen sobre gráficas fijas, pero que, salvo localmente, tienen una topología desconocida.

En la primera sección hablaremos un poco más sobre este tipo de algoritmos y sus particularidades, asi como los principales enfoques con los que suele encararse su diseño.

La segunda sección trata sobre algoritmos de ruteo local, esto es aquellos en los que cada nodo toma decisiones basándose solamente en información cercana. El propósito es mostrar que esto es posible, y al mismo tiempo describir herramientas útiles para el problema del Árbol de Steiner.

En la tercera sección se introducen los conceptos de k-generador y gráfica de Delaunay, y su relación con el problema. El punto principal aquí es la construcción local de un k-generador plano y la presentación del algoritmo que lo utiliza para aproximar el Árbol de Steiner.

Finalmente, en la cuarta sección presentamos algunas observaciones importantes sobre la naturaleza del algoritmo propuesto.

Nuestro objetivo en este capítulo es plantear los resultados de los capítulos anteriores en el mundo de las redes inalámbricas. Como consecuencia tendremos que introducir muchos conceptos nuevos. Por esta razón trataremos los temas más superficialmente que antes, pero al mismo tiempo proporcionaremos suficiente bibliografía que profundiza mucho más en cada uno.

5.1. Redes Móviles

Para los conceptos que manejamos en esta sección se recomienda consultar algún texto sobre sistemas distribuidos, por ejemplo [12].

Supongamos que queremos mandar un mensaje de un nodo a a otro b en una red inalámbrica cuyos nodos están en constante movimiento. En cada instante la red se puede representar mediante una gráfica como hemos hecho hasta ahora, pero a instantes distintos pueden corresponder gráficas distintas. Cualquier algoritmo que resuelva este problema debe decidir una ruta de la red por la cual mandar el mensaje. Por lo tanto el algoritmo tiene básicamente dos opciones: procurarse la información necesaria para establecer la forma que tiene la red en ese momento y en base a esto establecer la ruta que el mensaje debe tomar, o bien utilizar solo información local al decidir y olvidarse del resto de la red.

Como la primera opción parece la más simple, es la que exploraremos primero. Supongamos que existe un nodo especial con acceso directo a todos los demás, que a cada momento puede saber la forma que tiene la gráfica (por ejemplo un satélite muy poderoso). Con esta información podría recibir la petición de a de comunicarse con b y mandarle de regreso la ruta óptima para la configuración actual de la red. El nodo a cargaría esta información dentro del paquete que quiere mandar y luego lo soltaría a la red para que se llevaran a cabo sus instrucciones. En este modelo el único trabajo que realizan los nodos de la red es iniciar peticiones o reenviar mensajes, mientras que las decisiones de ruteo quedan todas a cargo del nodo especial. Por esta razón se le conoce como modelo centralizado.

Aun cuando no contemos con un nodo especial de este tipo, se puede aplicar el mismo estilo centralizado para resolver el problema. Por ejemplo, el nodo a puede inundar la red con mensajes pidiendo información sobre la configuración de cada nodo. Cuando un nodo recibe un mensaje de este tipo, junta la información local que posee de la gráfica en un nuevo mensaje, y transmite a su vez esta información a toda la red. Es bastante simple idear un mecanismo para asegurarnos que el intercambio de mensajes termine eventualmente, de forma que a tenga todos los pedazos y pueda reconstruir la gráfica completa. En este momento a funciona como el nodo central que calcula la ruta, la imprime en un nuevo paquete junto con el mensaje para b y lo suelta dentro de la red para que se sigan sus instrucciones.

Cuando se tiene la información completa de la gráfica, se puede utilizar cualquiera de los algoritmos de ruteo desarrollados para redes estáticas para resolver el problema del árbol de Steiner.

5.1.1. Algoritmos Distribuidos

La gran falla del modelo centralizado es precisamente que carga todo el trabajo en un solo nodo. Esto no solo nos obliga a averiguar la forma de la red antes de comenzar el envío mismo del mensaje, sino que si los nodos a y b van a establecer una comunicación durante un tiempo prolongado, el trabajo principal se concentrará en ellos dos (o en el nodo central), mientras que el resto de la red quedará sin trabajo por la mayor parte del tiempo. Los algoritmos distribuidos son una solución a este problema.

En este modelo cada nodo de la red tiene los mismos poderes que los demas, y cada uno hace una parte del cálculo de la solución del problema. Así por ejemplo, si se necesita calcular un Árbol Generador de Peso Mínimo, un algoritmo distribuido podría hacer que cada nodo calculara la parte del árbol que es incidente a él. Los distintos nodos se pasarían mensajes unos a otros hasta que todos se pusieran de acuerdo, y en este punto el árbol estaría calculado. A diferencia del enfoque centralizado, ni la gráfica completa ni el árbol mismo necesitan residir en un solo nodo, sino que cada uno almacena (y trabaja con) información parcial.

Cuando se distribuye el trabajo de esta manera debe existir cierta coordinación entre los nodos de la gráfica. Por ejemplo, debe ser posible decidir si un mensaje que se reciba es una respuesta a algún mensaje anterior o se trata de una nueva petición, o incluso si dos mensajes consecutivos llegaron en el orden que debían o no. Para esto se pueden poner restricciones al envío de mensajes de forma que la red esté *sincronizada*. El resultado efectivo de la sincronización es que un nodo puede establecer una línea de tiempo para los mensajes que recibe, i.e. hay un reloj lógico en la red que es respetado por todos los nodos, y con el cual se puede decidir el orden de los mensajes. Por supuesto, la sincronización impone un costo extra a las comunicaciones sobre la red.

La otra opción es que el sistema sea *asíncrono*, en cuyo caso realmente no se puede establecer un orden entre los mensajes. Recuérdese que estamos hablando de lo que cada nodo de la red ve, y que no tiene conocimiento global del sistema, por lo que si recibe dos mensajes del mismo origen no tiene manera de saber cuál se generó primero. En estas circunstancias el algoritmo debe tomar precauciones para poder interpretar lo que está pasando en cada situación posible.

En cualquier caso esta necesidad de coordinación puede ser costosa e inevitable. De hecho, por lo general la complejidad de los algoritmos distribuídos no se mide en el tiempo que tardan las cosas en calcularse, sino en el número de mensajes que deben pasar por la red para que los nodos se pongan de acuerdo en una solución. Esto tiene sentido por dos razones: la primera es que casi siempre la intensidad de los cálculos que debe realizar cada nodo se reduce mucho, pues el trabajo se reparte en toda la red; la segunda es que el tiempo que tarda un mensaje en pasar de un nodo a otro suele ser mayor que el que tarda un nodo en realizar sus cálculos. El cuello de botella en estos sistemas deja de ser la velocidad de procesamiento de los nodos, y se convierte en la velocidad de transmisión entre ellos.

Existen algoritmos distribuidos que resuelven el problema de encontrar las

distancias mínimas entre vértices (ya sea usando BFS, Dijkstra, o alguno de los otros algoritmos que hemos discutido), así como otros para resolver el problema del Árbol Generador de Peso Mínimo (consúltese [12], que también trata estos temas). Aplicando sus ideas y la teoría que discutimos en los capítulos anteriores sobre Árboles de Steiner, es posible crear algoritmos distribuídos para aproximar la solución como hicimos antes.

No profundizaremos más en esto, pues nuestro interés va dirigido a otra clase de algoritmos. Aunque los algoritmos distribuídos de este tipo reparten mejor la carga de trabajo entre los nodos de la red, tienen algunas desventajas. Por ejemplo, el envío de mensajes de a a b requiere que a inicie una petición en la red para que se calculen las estructuras de datos necesarias, y solo hasta que este proceso termine es que puede empezar el envío mismo del mensaje. Cuando el diámetro de la red es grande,[1] a puede tener que esperar un buen rato antes de tomar una decisión sobre hacia dónde debe mandar el mensaje (véase la figura 5.1). Mientras el algoritmo requiera la información completa de la distribución de la red para decidir sobre la ruta de envío, es probable que no podamos evitar esta espera.

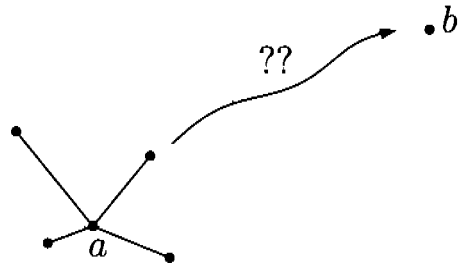


Figura 5.1: El nodo a quiere mandar un mensaje a b pero no conoce la forma de la gráfica, por lo que tiene que esperar información de otros nodos para poder decidir hacia dónde enviar el mensaje, o bien tomar una decisión localmente.

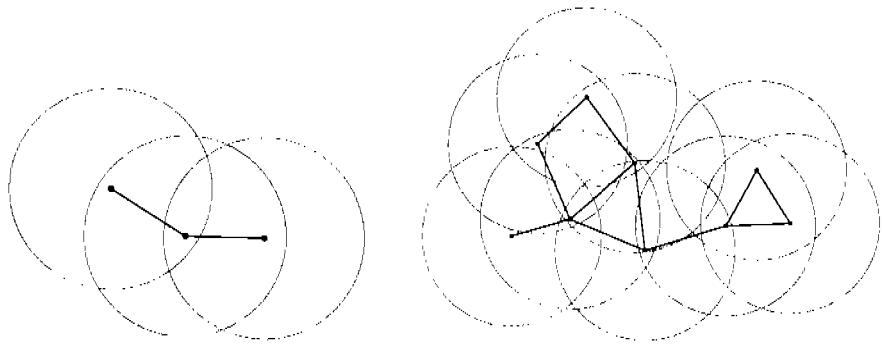
La alternativa que nos queda es olvidarnos de la topología de la gráfica, y decidir el siguiente paso de la ruta basados solamente en información local del nodo. De esta forma no es necesario esperar la respuesta de nodos lejanos para tomar decisiones. Hablaremos más sobre esto en la sección 5.2.

5.1.2. Gráficas de Disco Unitario

Estas gráficas son muy útiles para modelar cierto tipo de redes inalámbricas, como las redes de celulares. En abstracto, un celular es un dispositivo receptor con un pequeño procesador y un pequeño transmisor. La potencia de su transmisor es fija, y el celular solo puede mandar mensajes a otros dispositivos que se encuentren dentro de su radio de influencia (fig. 5.2). Si ponemos varios celulares dispersos en un área determinada, entre todos forman naturalmente una red

[1] El diámetro de una gráfica se define como la distancia máxima que hay entre dos vértices cualesquiera.

de comunicación. Suponiendo que todos los celulares tienen aproximadamente la misma potencia de transmisión, podemos normalizar esta distancia para que represente una unidad. De esta forma, dos nodos de la red se pueden comunicar si y solo si la distancia entre ellos es menor a 1.



(a) (b)

Figura 5.2: (a) La condición para que una arista pertenezca a la gráfica es que la distancia entre sus dos extremos sea a lo más 1. (b) Un ejemplo de gráfica de disco unitario.

5.2. Ruteo Local

Para estudiar las ideas principales involucradas en este tipo de ruteo presentaremos un par de algoritmos de ejemplo, que también nos serán de utilidad en nuestro estudio de Árboles de Steiner.

Algunos algoritmos distribuidos tienden a requerir que los nodos de la gráfica almacenen información útil para tomar decisiones de ruteo. Cuando el número de transferencias sobre la red se incrementa, la cantidad de información almacenada se acumula. Esta carga extra en los nodos puede ser inaceptable en ciertas aplicaciones. Cuando hablamos de ruteo local tenemos en mente dos puntos principales: el primero ya lo habíamos discutido, y es el de acelerar el proceso de toma de decisión de un vértice lo más posible, es decir, que un mensaje sea reenviado sin demora al destino; el segundo punto es que todo esto se lleve a cabo sin dejar huella sobre la red, lo cual significa que los nodos no almacenan información de ruteo (obviamente, es también inaceptable cargar esta información en el mensaje).

Con este modelo se tiene la ventaja de que los mensajes son despachados rápidamente, y cada nodo puede utilizar la memoria liberada para otras tareas. Suena muy bien, pero no es claro siquiera que se pueda diseñar algo útil con estas restricciones. Surge la pregunta de si es posible asegurar que un mensaje llegará a su destino si solo se rutea localmente. Nuestro siguiente objetivo es mostrar que si es posible.

Hay que observar que hay gráficas en las que todas estas restricciones hacen imposible el ruteo. Por ejemplo, en gráficas abstractas donde no hay más información que las aristas y vértices que la componen, para un vértice que solo cuenta con información local todas sus aristas incidentes lucen igual. No hay punto de comparación para distinguirlas. Cualquier algoritmo determinista decidirá sobre alguna de las aristas y enviará el mensaje por allí, después de lo cual olvidará completamente que ha visto el mensaje y por dónde lo mandó (pues está prohibido que registre esta información). El resultado es que el mensaje se podría quedar dando vueltas en la gráfica indefinidamente sin llegar a su destino jamás.

Por esta razón, permitiremos que los mensajes tengan una cantidad constante (pequeña) de memoria extra disponible, donde puedan almacenar información importante (como la identidad del vértice de destino, o de algún vértice por el que ya pasaron). Esto va de acuerdo al paradigma local, pues en ningún momento se tiene información sobre la distribución completa de la red, sino solamente de algunos de sus elementos. De ahora en adelante asumiremos que la gráfica está dibujada en el plano, y que cada vértice conoce sus coordenadas en 2D. Esta información geométrica le permitirá a los algoritmos orientarse al tomar sus decisiones.

5.2.1. Ruteo con Brújula

Este algoritmo y el que sigue fueron propuestos por Kranakis *et al.* [10] La idea es la más intuitiva que hay: si se conoce la dirección en la que queda el destino, hay que tomar la ruta cuya dirección sea lo más directa posible. Esto es lo que podría hacer una persona en París para llegar a la Torre Eiffel, caminar hacia donde se vea la punta de la torre.

Suponiendo que un nodo tiene la tarea de reenviar un mensaje y solo cuenta con las coordenadas del destino, el algoritmo trabaja como sigue. Averigua primero las coordenadas de todos sus vecinos inmediatos. Con base en esta información, calcula las pendientes de sus aristas incidentes y calcula también la pendiente de la recta que pasa por sus propias coordenadas y las del destino. Finalmente escoge la arista cuya pendiente se aproxime más a la de la recta y envía el mensaje por allí. Si llegan a haber empates, elige aleatoriamente cualquiera de las aristas empatadas. Cada vértice de la gráfica ejecuta el mismo algoritmo hasta que el mensaje se encuentre en su destino.

Está claro que el algoritmo cumple las condiciones de localidad que mencionamos anteriormente, pues en ningún momento almacena información (ni en el nodo, ni en el mensaje) y su decisión está basada solamente en su vecindad inmediata. Esta es una propiedad con repercusiones importantes: una vez que el mensaje es enviado, tanto el nodo como el mensaje se olvidan de la existencia del otro.

Desgraciadamente, esta ingenuidad del algoritmo lo hace cometer errores en ocasiones. Hay gráficas (fig. 5.3) en las que el mensaje nunca llega a su destino.

Una buena pregunta es si podemos decir algo sobre el tipo de redes en las cuales el algoritmo funciona. En particular, la Gráfica de Delaunay es un ejemplo

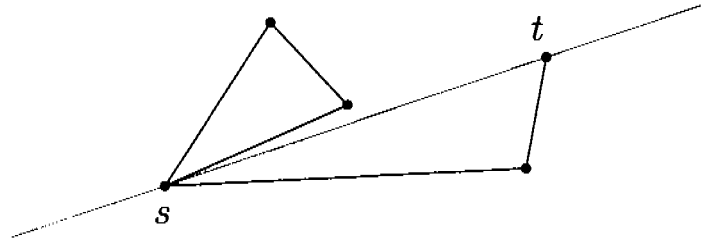


Figura 5.3: En esta gráfica el Ruteo con Brújula falla al intentar enviar un mensaje de s a t.

en el cual cada vez que un nodo reenvía el mensaje, la distancia entre éste y su destino decrece. Dado que la gráfica es finita, esto significa que eventualmente el mensaje será recibido correctamente. Hablaremos un poco más sobre esta gráfica, que tiene propiedades muy interesantes, cuando lleguemos a la siguiente sección.

5.2.2. Ruteo por Caras

Este algoritmo es similar al ruteo con brújula, pero funciona en más gráficas. Se trata también de un algoritmo local y para cualquier gráfica plana garantiza que el mensaje ruteado llegará a su destino.

Cuando una gráfica es plana sus aristas forman una partición del plano en regiones, a las que les llamaremos caras. El segmento de recta que atraviesa los nodos origen y destino de un mensaje cruza estas caras en orden, por lo que si dos caras son cruzadas una después de la otra, necesariamente deben tener alguna arista en común que además intersecte al segmento. Partiendo del origen, se puede seguir el contorno de la primera cara hasta intersectar la recta. En ese momento cambiamos y comenzamos a recorrer la cara adyacente, y así sucesivamente hasta llegar al destino. Esta es la idea general del algoritmo.

Para implementarlo localmente hay que tener algunas cosas en cuenta. Lo primero es que hay que recordar las coordenadas del origen y del destino para poder calcular la recta guía. También, dos caras pueden compartir más de una arista, por lo que para evitar problemas hay que hacer el cambio de caras en el punto más lejano al origen que sea posible. De esta forma se evita repetir caras en el recorrido. Para esto es necesario almacenar otros dos datos en el mensaje: el vértice donde comenzó el recorrido de la cara actual, y el punto de intersección con la recta guía más lejano al origen que se haya encontrado.

Estos datos se pueden actualizar fácilmente usando la información geométrica local de cada nodo, lo cual sugiere que se realice el ruteo mediante los pasos que se explican a continuación. Primero, registrar el nodo de partida y comenzar a recorrer la primera cara en una dirección fija, arista por arista. Cada vez que el mensaje pase por una arista que cruce al segmento de recta guía determinado por el origen y el destino, comparar la distancia entre la intersección y el nodo origen, y si es la mayor encontrada hasta el momento registrar el punto. Cuando

el mensaje regrese al vértice del que partió, solo necesita volver a recorrer la cara hasta la posición correcta y hacer el cambio de cara. Se repite el proceso hasta encontrar el vértice de destino.

La idea es similar al Ruteo con Brújula, excepto que en vez de rutear el mensaje nodo por nodo, se hace cara por cara. Con este cambio es posible asegurar que el mensaje siempre se entrega a su destino usando solo operaciones locales, siempre y cuando la gráfica sea plana.

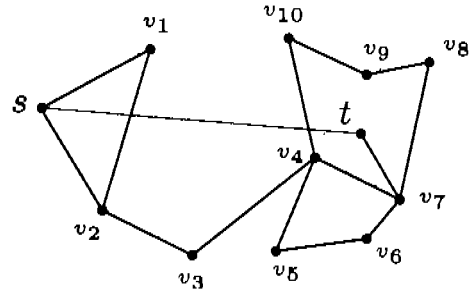


Figura 5.4: Un ejemplo del Ruteo por Caras. En esta gráfica s quiere mandar un mensaje a t, por lo que siguiendo el algoritmo primero recorre la cara sv_1v_2s, cambia a la cara exterior en la arista (v_1, v_2), la recorre y vuelve a cambiar de cara en la arista (v_4, v_{10}), continua recorriendo $v_4v_{10}v_9v_8v_7t$, y en este momento se detiene pues ya llegó a su destino. El segmento $\overline{st}$ no pertenece a la gráfica, solo aparece en la figura como referencia.

Obsérvese que en la discusión de los dos últimos algoritmos no hemos hablado sobre trayectorias mínimas. La ruta que toma el mensaje para llegar a su destino puede no ser óptima. Lo que hemos mostrado hasta ahora es que el ruteo se puede llevar a cabo usando solo información local y una cantidad constante de memoria en el mensaje.

5.2.3. Aproximación Local al MST

Cuando la gráfica no es plana necesitamos primero encontrar una subgráfica que sí lo sea, de forma que podamos utilizar el Ruteo por Caras. Puesto de otra manera, tenemos que poder identificar localmente a las aristas que pertenecen a una subgráfica plana y las que no, para basar las decisiones de ruteo solamente sobre estas. No solo eso, sino que hay que hacerlo de forma que los distintos nodos estén de acuerdo a este respecto, en el sentido de que los dos extremos de una arista decidan exactamente lo mismo sobre su pertenencia en la subgráfica.

Esta es una idea poderosa, y el algoritmo de Chávez *et al.* [3] para gráficas de disco unitario es una muestra de que es una idea factible. La subgráfica que construye tiene muchas propiedades muy interesantes, una de las cuales es que dado un número $t \geq 2$, su peso es a lo más $\frac{t+1}{t-1}$ veces el peso del Árbol Generador de Peso Mínimo de la gráfica. La construcción es local, y requiere solamente que cada vértice conozca su t-vecindad (los vértices que se pueden alcanzar desde el

vértice saltando t aristas o menos).

El proceso es bastante directo. Lo primero que hay que hacer es establecer un orden total entre las aristas de la gráfica de forma que el Árbol de Peso Mínimo sea único. Esto es lo que permite que los nodos estén de acuerdo en cuales aristas seleccionar y cuales no. Una vez hecho esto, cada vértice encuentra su t-vecindad y le construye un Árbol Generador de Peso Mínimo. Aquellas aristas que pertenezcan al "árbol local" de cada uno de sus extremos van dentro de la solución, y el resto son descartadas.

Tanto el algoritmo como la gráfica son bastante impresionantes por su simplicidad y por las propiedades que poseen. Usando este método, con solo analizar la 3-vecindad de cada vértice se puede construir localmente una 2-aproximación al Árbol Generador de Peso Mínimo de la red. Aunque es una buena aproximación, no nos sirve directamente para el problema del Árbol de Steiner, pues para poder explotar las ideas de los capítulos anteriores lo que realmente tendríamos que hacer es buscar una manera de construir localmente el Árbol Generador de Peso Mínimo de la gráfica de proximidad de los vértices terminales. Sin embargo ideas introducidas por el algoritmo nos serán de utilidad en la siguiente sección.

Incluso si resolviéramos este problema, faltaría mostrar un algoritmo de ruteo que asegurara una entrega cercana a la óptima para cada paquete. Volveremos a este punto más adelante.

5.3. k-Generadores

Volviendo al problema de los árboles de Steiner, sea G una gráfica y S un conjunto de vértices terminales. Digamos que existe un algoritmo de ruteo que asegura que para cualquier pareja de vértices puede hacer llegar localmente un mensaje de uno a otro usando una ruta cuyo peso sea a lo más k veces el de la ruta óptima. Al parámetro k se le conoce como *factor de estiramiento*. Si un vértice de S quisiera enviar un mensaje al resto de los terminales y conociera la distancia en G a la que se encuentran, podría calcular la gráfica de proximidad para S y encontrarle un Árbol Generador de Peso Mínimo. Usando esta información y el algoritmo de ruteo, podría llevarse a cabo la distribución del mensaje de forma que la distancia total recorrida fuera a lo más $2k$ veces la óptima, de acuerdo al teorema 3.1.

Definición 5.1. Si G es una gráfica y H una subgráfica de G, se dice que H es un *k-generador* de G si y solo si contiene a todos sus vértices, y para todo par de vértices u, v

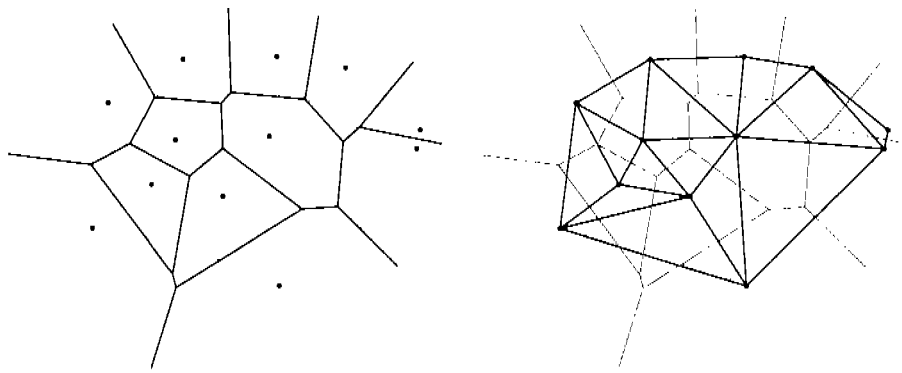
$$\delta_H(u, v) \leq k \cdot \delta_G(u, v)$$

Toda trayectoria mínima en un k-generador tiene factor de estiramiento k. En esta sección estudiaremos la construcción local de k-generadores en redes móviles.

5.3.1. Gráfica de Delaunay

Existen muchas disciplinas que han llegado independientemente a los conceptos de Diagrama de Voronoi y Gráfica de Delaunay [1, 13] de un conjunto de puntos. Estas estructuras están íntimamente relacionadas y poseen una riqueza de propiedades comparable con pocas otras.

Para un conjunto P de puntos en el plano, su *Diagrama de Voronoi* es una partición del plano en regiones convexas. Cada región es generada por un elemento p del conjunto y consiste de todos aquellos puntos del plano que son más cercanos a p que a cualquier otro elemento.



(a) Diagrama de Voronoi (b) Gráfica de Delaunay

Figura 5.5: Ejemplo de Diagrama de Voronoi y su dual, la Gráfica de Delaunay.

Definición 5.2. Sea G una gráfica plana. La *gráfica dual* de G es una gráfica que contiene un punto por cada cara de G y una arista entre dos puntos si y solo si las caras respectivas son adyacentes.

La Gráfica de Delaunay se define como la dual del Diagrama de Voronoi. Como a cada región del diagrama corresponde un solo elemento de P, la gráfica de Delaunay se puede construir sobre estos puntos de forma que dos son adyacentes si y solo si sus regiones de Voronoi respectivas son adyacentes. En la figura 5.5 se ilustran ambas estructuras. La gráfica de Delaunay es plana, conexa, y es fácil ver que cuando no hay 4 elementos de P co-circulares todas sus caras son triángulos. En este caso se puede dar una definición alterna para la gráfica, que consiste de todos aquellos triángulos formados con puntos de P cuyo circuncírculo no contenga ningún otro elemento de P.

Obsérvese que la gráfica de Delaunay se define sobre un conjunto de puntos, no sobre una gráfica. Keil *et al.* [9] demostraron que la gráfica de Delaunay es un k-generador de la gráfica euclideana completa con conjunto de puntos P, donde $k = \frac{4\sqrt{3}}{9}\pi \approx 2{,}42$. Esto es, la distancia entre dos vértices en la gráfica de Delaunay es a lo más 2,42 veces la distancia euclideana entre ellos.

Las propiedades de la gráfica de Delaunay son excelentes para usarla como red subyacente en el problema del Árbol de Steiner: el ruteo con brújula siem-

pre es posible, sus distancias tienen factor de estiramiento 2,42 (respecto de la distancia euclideana), y es plana, lo cual simplifica el ruteo. La clave radica en poder identificar sus aristas localmente.

5.3.2. Construcción Local de un k-Generador

La gráfica de Delaunay puede tener aristas arbitrariamente largas, por lo que usando solamente aristas de una gráfica de disco unitario, cuya longitud a lo más 1, no siempre es posible construirla en un conjunto de puntos dado. En otras palabras, al concentrar nuestro estudio en gráficas de disco unitario estamos eliminando la posibilidad de utilizar la teoría general desarrollada para la gráfica de Delaunay.

Se puede definir una versión restringida de la gráfica en la cual se eliminan todas las aristas con longitud mayor a 1. Li *et al.* [11] demostraron que la gráfica de Delaunay restringida de esta manera es un $\frac{4\sqrt{3}}{9}\pi$-generador de la gráfica de disco unitario, y describen un algoritmo para construir localmente una gráfica plana que la contiene. Explicaremos su construcción a continuación.

Dado un triángulo en la gráfica de disco unitario, es posible que los vértices que lo forman no puedan decidir localmente si se trata de una cara de la triangulación de Delaunay. Por ejemplo, si el triángulo tiene algún ángulo interior muy grande, puede existir un vértice muy lejano que esté contenido dentro de su circuncírculo. Por esto y dado que queremos un algoritmo local, en vez de construir la gráfica de Delaunay restringida mostraremos como calcular una gráfica parecida que la contiene.

La *Gráfica de Delaunay Local* (GDL) se define sobre una gráfica de disco unitario en base a un parámetro t. Contiene a todas las aristas de longitud a lo más 1 que caigan en alguna de las siguientes categorías (fig. 5.6)

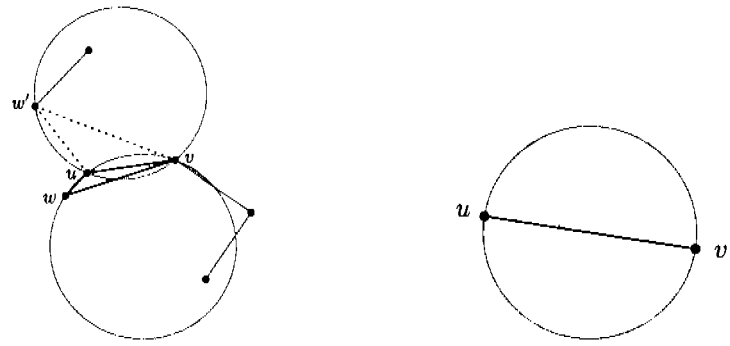
1. Aquellas aristas que sean parte de algún triángulo $\triangle uvw$, cuyo circuncírculo no contenga en su interior vértices de la t-vecindad de u, v o w.

2. Cualquier arista tal que el círculo que la tiene como diámetro no contenga en su interior otros vértices de la gráfica.

GDL contiene a la gráfica de Delaunay restringida, por lo que también es un k-generador de la gráfica de disco unitario. Aunque no necesariamente es plana cuando $t = 1$, tiene una cantidad lineal de aristas y se puede hacer plana eficientemente preservando sus propiedades.

Su construcción es local siguiendo un método parecido al de la aproximación a MST que presentamos antes. Primero, cada vértice se procura la información de la subgráfica inducida por su t-vecindad, para la cual construye una gráfica de Delaunay. Luego prueba cada arista para ver si cae en alguno de los dos casos mencionados. Sus candidatos son enviados a sus vecinos inmediatos, y si estos los aceptan también, las aristas entran a la gráfica, si no son descartadas.

Con esta herramienta podemos proponer el siguiente algoritmo para aproximar el Árbol de Steiner en una gráfica de disco unitario utilizando solamente



(a) Aristas de Triángulos de Delaunay Locales (b) Aristas de Gabriel

Figura 5.6: (a) Cuando $t = 1$, las aristas (u, w') y (v, w') no pertenecen a GDL, pues dentro del circuncírculo de $\triangle uvw'$ hay un nodo en la 1-vecindad de w'. Sin embargo, el triángulo $\triangle uvw$ si pertenece a GDL.
(b) Segunda categoría. Todas las llamadas *aristas de Gabriel* pertenecen a GDL.

información local. Suponemos que las coordenadas de todos los vértices terminales son conocidas para el vértice que inicia el ruteo. Primero plantearemos el algoritmo y después discutiremos con más detalle algunas de sus partes.

1. Si solo hay un vértice terminal, no hay nada que rutear y el algoritmo termina.

2. El vértice inicial utiliza la información de los vértices terminales que conoce para construir una gráfica de proximidad a partir de las distancias *euclideanas* entre ellos.

3. Calcula el MST de esta gráfica, lo cual le indica las aristas que debe seguir para realizar el ruteo. Cada una de estas aristas representa un camino de la gráfica de disco unitario por el cual se debe enviar el mensaje.

4. Por cada rama del MST que incide en el vértice se envía un paquete que incluya el mensaje y la información de los vértices terminales que pertenecen a esa rama. De esta forma el conjunto de vértices terminales de destino queda particionado entre los paquetes: la información de cada vértice terminal de destino se encuentra en uno y solo uno de los paquetes enviados.

   Para enviar cada paquete se utiliza el ruteo por caras o algún algoritmo similar sobre la Gráfica de Delaunay Local.

5. Una vez que un paquete llega a un vértice terminal, se convierte en el nuevo vértice inicial y se repite el algoritmo con el subconjunto de vértices terminales pertinente.

Los pasos 3 y 4 del algoritmo imponen un orden en la manera en que los vértices terminales reciben el mensaje. Aquellos que son incidentes al vértice inicial en el MST lo reciben primero y eventualmente lo reenvian a los nodos que faltan. Hacemos esto para poder explotar los resultados sobre la 2-aproximación que presentamos anteriormente.

Existe un error sutil pero importante en este algoritmo. En el paso 2 utilizamos la distancia euclideana para construir la gráfica de proximidad, cuando la distancia que deberíamos utilizar es la de la gráfica de disco unitario sobre la que estamos trabajando. Sin embargo no tenemos manera de conocer esta última distancia sin antes recorrer la gráfica, o bien sin que exista algún oráculo que nos proporcione esta información. Como queremos un algoritmo distribuido local, debemos evitar estas situaciones, por lo que utilizamos la heurística de tomar la distancia euclideana como una aproximación a la distancia en la gráfica.

En general estas dos distancias no estan relacionadas, y no es difícil construir un ejemplo de gráfica de disco unitario en la que dos vértices cuya distancia euclideana sea $1 + \epsilon$, en la gráfica estén arbitrariamente alejados. Esto anularía nuestros resultados, pues la información que usamos para construir la aproximación al Árbol de Steiner no refleja la situación de los nodos en la gráfica. Por ahora ignoraremos este problema, pero regresaremos a el más adelante.

Experimentalmente, el ruteo con brújula sobre GDL es bastante eficiente, pues se obtiene en promedio un factor de estiramiento menor a 2. Por tanto, usando estas técnicas para el problema del Árbol de Steiner en redes móviles (suponiendo que son modelables usando gráficas de disco unitario) la aproximación esperada que se obtiene con el algoritmo es 4 veces más pesada que el Árbol de Steiner real. Este análisis se basa solamente en resultados de experimentos, por lo que no es un resultado formal. Para esto necesitaríamos un algoritmo que asegure rutear localmente un mensaje con un factor de estiramiento fijo, que hasta ahora no hay.

5.4. Observaciones Finales

La razón por la que la distancia en una gráfica de disco unitario puede ser mucho mayor que la distancia euclideana es que los caminos tienen que rodear los "agujeros" que se generan por la topología de la gráfica. Un agujero es realmente una región cerrada y conexa del plano delimitada por aristas de la gráfica, y que no es atravesada en su interior por ninguna arista, i.e. no hay aristas que sirvan de puentes para ir de un lado al otro de la región. La única manera de rutear a través de un agujero es rodeándolo, lo cual incrementa la distancia que se recorre en la gráfica, mientras que la distancia euclideana permanece fija.

Para que el algoritmo propuesto en la sección anterior sea de utilidad, necesitamos alguna condición que nos asegure que la distancia euclideana aproxima bien a la distancia en la gráfica. Aunque pudiera parecer que esto es pedir demasiado, en las redes móviles del mundo real no es una condición extraña. Intuitivamente, si los nodos de la red están distribuidos uniformemente en el plano

con una densidad suficiente, la probabilidad de que existan agujeros grandes se ve muy reducida, por lo que en la mayoría de los casos el comportamiento del algoritmo será cercano al óptimo.

Por último obsérvese que en este caso el cálculo del Árbol Generador de Peso Mínimo de la gráfica de proximidad solo nos sirve para orientarnos hacia la manera en que conviene comenzar el ruteo. El ruteo mismo sobre la gráfica de disco unitario se puede hacer de muchas maneras, incluso quizá explotando la heurística del conjunto fundamental, de forma que si en el proceso de ruteo el algoritmo se da cuenta que le conviene dividir el paquete aun cuando no se encuentre en un vértice terminal, lo pueda hacer. Valdría la pena estudiar un poco mas a fondo esto como una posible optimización.

Capítulo 6

Conclusiones

Dado que el problema general del Árbol de Steiner es NP-completo, es muy probable que lo mejor que podamos hacer para atacarlo es encontrar una buena aproximación.

La idea fundamental que utilizamos en nuestras aproximaciones es la de calcular el Árbol Generador de Peso Mínimo de una gráfica relacionada, la gráfica de proximidad, que es construída con información de las distancias entre los vértices de la gráfica original. Esta idea permite encontrar una 2-aproximación sencilla que en la práctica funciona bastante bien, y que en la teoría resulta difícil de mejorar. Aun así existen otros algoritmos mas complejos para los cuales se puede probar una mejor aproximación. La ventaja de la 2-aproximación radica en que es una composición de otros algoritmos muy estudiados, lo cual permite escalar con relativa facilidad sus resultados a algoritmos de ruteo distribuido.

La relación profunda que existe entre los Árboles de Steiner y los Árboles Generadores de Peso Mínimo nos permite entender con mayor claridad la manera en que esta aproximación funciona. En particular hemos observado que si logramos identificar los posibles vértices de ramificación del Árbol de Steiner al construir la aproximación, los resultados que obtendremos serán mejores. Este resultado sirve como inspiración para diseñar heurísticas que mejoran el comportamiento del algoritmo, y se puede resumir de la siguiente manera: si podemos estar seguros que un vértice no-terminal es un vértice de ramificación del Árbol de Steiner que buscamos, lo que obtengamos al añadirlo al conjunto de vértices terminales y construir la 2-aproximación será mas cercano al Árbol de Steiner real. En este sentido, un resultado teórico interesante sería una heurística eficiente para la cual se haya probado que siempre mejora la aproximación en el peor caso.

El paradigma de ruteo local apenas comienza a estudiarse, pero ya ha arrojado algunos resultados impresionantes, de los cuales los que presentamos en el capítulo anterior son solo una muestra. En esta área falta encontrar un algoritmo que asegure un factor de estiramiento constante bajo para rutear localmente un mensaje entre cualquier pareja de vértices, lo cual sería una herramienta muy útil para muchos otros algoritmos. Por lo pronto una buena manera de resolver

este problema es utilizando algoritmos de ruteo locales (como el ruteo con brújula) sobre una subgráfica que tenga propiedades útiles, y que también se pueda construir de forma local. En las gráficas de disco unitario, el ruteo con brújula sobre un k-generador como la Gráfica de Delaunay Local funciona bastante bien en la práctica.

Cuando podemos determinar estáticamente (i.e. sin la ayuda de otros nodos) la distancia aproximada a la que se encuentran los nodos terminales en una gráfica de disco unitario, podemos aplicar todas estas ideas para obtener una buena aproximación al Árbol de Steiner con métodos puramente locales.

Alternativamente, una pregunta abierta es la de si existe algún algoritmo que pueda identificar localmente las aristas de una aproximación al Árbol de Steiner y sobre estas realizar el ruteo. Este acercamiento es distinto al que tomamos nosotros, y quizá se podrían utilizar las mismas técnicas que hay para la construcción local de otras gráficas. De lograrse, probablemente se eliminaría el uso de los k-generadores, e incluso posiblemente la necesidad de aproximar la distancia entre los vértices terminales.

Para el caso de las redes que se pueden modelar usando gráficas de disco unitario, habría que buscar algoritmos que trataran de minimizar la distancia topológica (*en saltos*) entre sus vértices, para que el mensaje ruteado sea reenviado el menor número de veces posible, lo cual puede ser mas importante que minimizar la distancia total recorrida. El problema principal que se presenta al cambiar la función de distancia dentro de la gráfica es que no la podamos aproximar estáticamente para construir una gráfica de proximidad fiable. Por esto, el caso general para distancias arbitrarias sigue siendo un problema abierto.

Apéndice A

Definiciones Básicas de Teoría de Gráficas

A continuación se presentan algunas definiciones de conceptos que utilizamos en la obra. Para una discusión más avanzada de este material se recomienda consultar la bibliografía proporcionada.

Definición A.1. Sea V un conjunto cualquiera y E un conjunto de parejas de elementos de V. Decimos que $G = (V, E)$ es una *gráfica*, cuyos *vértices* y *aristas* son los elementos de V y E, respectivamente. Si $u = \overline{xy}$ es una arista de la gráfica, decimos que u *incide* en los vértices x y y.

Definición A.2. El *grado* de un vértice en una gráfica se define como el número de veces que las aristas de la gráfica inciden en el.

Conceptualmente representamos a los vértices de la gráfica como puntos en un plano y a las aristas como líneas que los unen por parejas. Esta visualización es muy útil y ampliamente utilizada.

Cuando los elementos de E son parejas ordenadas decimos que la gráfica es *dirigida*, lo cual representamos mediante flechas en vez de líneas. Si no son parejas ordenadas entonces decimos que la gráfica es *no-dirigida*. A menos que indiquemos lo contrario se deberá asumir que las gráficas que utilizamos son no-dirigidas.

Definición A.3. Sea $G = (V, E)$ una gráfica. Decimos que $G' = (V', E')$ es una *subgráfica* de G cuando $V' \subset V$ y $E' \subset E$. Si además se tiene que $V' = V$ entonces decimos que G' es *generadora* de G.

Las subgráficas nos dan información parcial de la gráfica. A veces esta información es lo único que nos interesa de la gráfica, por lo que una subgráfica es una manera mas compacta de decir que una gráfica tiene alguna propiedad. Es el caso de las subgráficas generadoras, que nos dicen todos los vértices que pertenecen a la gráfica.

Definición A.4. Sean $G = (V, E)$ una gráfica y $\{v_0, v_1, \ldots, v_n\} \subset V$ tales que cada arista de la forma $\overline{v_i v_{i+1}}$ esté en E para $0 \leq i < n$. Entonces decimos que $P = (v_0, v_1, \ldots, v_n)$ es un *camino* en la gráfica G cuya *longitud* es n.

Al vértice v_0 se le llama *vértice inicial* del camino y a v_n *vértice final.*

Los caminos se visualizan como sucesiones de aristas en la gráfica, que se pueden recorrer yendo de un vértice a otro en orden. Los vértices de un camino están *conectados* por el camino. La longitud es el número de aristas que el camino recorre.

Definición A.5. Decimos que una gráfica es *conexa* cuando para cualquier pareja de sus vértices existe algún camino que los conecta. Si no es el caso decimos que la gráfica es *disconexa.* Las *componentes conexas* de una gráfica son sus subgráficas conexas maximales.

Cuando una gráfica es disconexa, su representación en el plano está compuesta de varios fragmentos. Cada fragmento es conexo pero no hay aristas entre fragmentos. Estos fragmentos son las componentes conexas.

Definición A.6. Una *trayectoria* es un camino que no repite vértices.

Definición A.7. Un *ciclo* es un camino en donde el vértice inicial es el mismo que el vértice final y ningún otro vértice se repite.

Definición A.8. La *distancia* entre dos vértices se define como el mínimo de las longitudes de los caminos que conectan a dichos vértices. Si no hay ningún camino que los conecte, la distancia entre dos vértices queda indefinida y en ocasiones se representa por el símbolo ∞.

Podemos asignar *peso* a las aristas de una gráfica. Es decir, definimos una función w que a cada arista le asocie un número, y añadimos esta función a la definición de la gráfica, de forma que $G = (V, E, w)$. Las gráficas "sin peso" son realmente un caso especial en el que el peso de todas las aristas es 1. Así pues, se puede redefinir la *longitud* de un camino como la suma de los pesos de sus aristas, y la definición de distancia se mantiene, pero ahora usando este nuevo concepto de longitud. Como un abuso de notación, nos referimos al peso de una gráfica como la suma de los pesos de sus aristas.

Es sencillo darse cuenta que cuando los pesos de las aristas son todos positivos, el camino cuya longitud corresponde a la distancia entre dos vértices es realmente una trayectoria, por lo que en este caso se pueden utilizar ambos conceptos de forma intercambiable.

Definición A.9. Un *árbol* es una gráfica conexa sin ciclos.

Finalmente, juntando varios de estos conceptos, nos referimos a un *árbol generador de peso mínimo* de una gráfica $G = (V, E, w)$ como una subgráfica generadora de G, conexa, sin ciclos y que es mínima en peso respecto a w.

Bibliografía

[1] Franz Aurenhammer, *Voronoi diagrams – a survey of a fundamental geometric data structure*, ACM Comput. Surv. (New York, NY, USA), vol. 23-3, ACM, 1991, págs. 345–405.

[2] Cüneyt F. Bazlamaçci y Khalil S. Hindi, *Minimum-weight spanning tree algorithms a survey and empirical study*, Comput. Oper. Res. (Oxford, UK, UK), vol. 28, Elsevier Science Ltd., 2001, págs. 767–785.

[3] Edgar Chávez, Stefan Dobrev, Evangelos Kranakis, Jaroslav Opatrny, Ladislav Stacho y Jorge Urrutia, *Local Construction of Planar Spanners in Unit Disk Graphs with Irregular Transmission Ranges*, LATIN, 2006, págs. 286–297.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein, *Introduction to Algorithms*, 2a. ed., The MIT Press, 2001.

[5] Reinhard Diestel, *Graph Theory*, 3a. ed., Graduate Texts in Mathematics, vol. 173, Springer, julio 2005.

[6] Michael R. Garey y David S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[7] Frank Harary, *Graph theory*, Addison-Wesley, 1969.

[8] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (New York) (R. E. Miller y J. W. Thatcher, eds.), Plenum Press, 1972, págs. 85–103.

[9] J. Mark Keil y Carl A. Gutwin, *The Delauney Triangulation Closely Approximates the Complete Euclidean Graph*, WADS '89: Proceedings of the Workshop on Algorithms and Data Structures (London, UK), vol. 382, Springer-Verlag, 1989, págs. 47–56.

[10] Evangelos Kranakis, Harvinder Singh y Jorge Urrutia, *Compass Routing on Geometric Networks*, Proc. 11 th Canadian Conference on Computational Geometry (Vancouver), August 1999, págs. 51–54.

[11] X. Li, G. Calinescu y P. Wan, *Distributed construction of a planar spanner and routing for ad hoc wireless networks*, IEEE INFOCOM, 2002.

[12] Nancy A. Lynch, *Distributed Algorithms*, 1a. ed., The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, 1997.

[13] F. P. Preparata y M. I. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, New York, 1985.

[14] Steven S. Skiena, *The Algorithm Design Manual*, Springer-Verlag, New York, 1997.