

Teoría de Algoritmos I

Joaquín Blanco. 30 de agosto de 2019

Teoría de Algoritmos I

El algoritmo de Gale-Shapley: Una introducción al análisis de algoritmos.

Bases del análisis de algoritmos.

Teoría de grafos.

Grafos dirigidos acíclicos y el orden topológico.

Prueba de bipartidismo: una aplicación de búsqueda en amplitud.

Algoritmo de Kosaraju para componentes fuertemente conexas.

Algoritmos Greedy.

Elementos de una estrategia greedy

Propiedad de elección codiciosa.

Subestructura optima.

Interval Scheduling.

Video Stream

The Minimum Spanning Tree

Códigos Huffman y compresión de datos

Caminos mínimos en un Grafo.

Mochila Fraccionaria.

The Blair Witch Project

The Security company.

Las propiedades de los arboles de recubrimiento mínimo

División y Conquista.

El algoritmo Mergesort.

Counting Inversions

Puntos extremos de un polígono convexo.

La mediana de dos conjuntos de elementos separados.

Finding the Closest Pair of Points.

The maximum-subarray problem.

Arreglos unimodales.

Invertir en acciones

Programación Dinámica.

Elementos de la Programación Dinámica.

Subestructura optima.

Sub problemas superpuestos

Reconstruyendo una solución optima.

El problema del corte de varillas.

Weigthed Interval Scheduling

Publicidad en carretera

El Problema del Viajante de Comercio o TSP

Cambio mínimo en monedas

Flujo de redes.

Marco teórico para el Flujo de Redes.

Variantes de Flujo de Redes.

El método de Ford-Fulkerson.

Redes residuales.

Aumentando caminos.

Corte de redes de flujo

El algoritmo básico de Ford-Fulkerson

Análisis del algoritmo de Ford-Fulkerson.

Bipartite Matching Problem

Diseño de encuestas

Selección de proyectos

Posible ganador en torneo

Segmentación de imágenes

Programación de vuelos

Complejidad NP

Complejidad NP y las clases P y NP.

Resolución en tiempo polinomial.

Verificación en tiempo polinomial.

Reducciones polinomiales.

Intractabilidad - co-NP

NP-Completo

Nuestro primer problema NP-Completo: SAT

Cobertura de vértices

Cobertura de conjuntos

Clique

Subset Sum

Problema del viajante

El algoritmo de Gale-Shapley: Una introducción al análisis de algoritmos.

$$f(x) = x^2$$
$$\text{Find the derivative}$$
$$pe(S) = \frac{y_1 - y_0}{x_1 - x_0} = \frac{g(x+h) - g(x)}{(x+h) - x} = \frac{g(x+h) - g(x)}{h}$$
$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$
$$f(x) = \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{2xh + h^2}{h}$$
$$= 2x$$
$$\text{Slope}(T) = \lim_{h \rightarrow 0} \frac{g(x+h) - g(x)}{h}$$
$$= \lim_{h \rightarrow 0} \frac{h(2x+h)}{h}$$
$$= \lim_{h \rightarrow 0} (2x+h)$$
$$\frac{df}{dx} \quad \left[\frac{d}{dx}(x^n) = nx^{n-1} \right] = \lim_{h \rightarrow 0} \frac{h(2x+h)}{h}$$
$$= \lim_{h \rightarrow 0} (2x+h)$$
$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$
$$f(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$
$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

Motivación.

Para comenzar en el mundo de la teoría de algoritmos, tomaremos un caso emblemático como lo es el del *Algoritmo de Gale-Shapley* para estudiarlo a fondo y determinar no solamente su orden de ejecución, sino tratar de entender los corolarios del mismo y las demostraciones de estos a fin de comenzar con una práctica que será más que habitual en lo que respecta al análisis de algoritmos. Sin más que decir, comencemos.

El problema.

El problema de los matching estables vio su origen en el año 1962, cuando David Gale y Lloyd Shapley se hicieron la siguiente pregunta: *¿Se puede diseñar un proceso de reclutamiento para una empresa, que se administre a sí mismo?*

Vamos a tratar de aclarar el panorama del problema. En principio, existen dos grupos (en iguales cantidades); las empresas, y los aspirantes. Cada empresa tiene un grado de preferencia para cada aspirante (en principio, distintos para cada uno de ellos) y de la misma forma, cada aspirante tiene un grado de preferencia para cada empresa (distinto para cada empresa). Basados en esto, cada empresa decide convocar a cada aspirante hasta encontrar a alguien que acepte trabajar con ellos. Los aspirantes pueden aceptar la invitación o no, dependiendo de si ya se encuentran ligados a otra empresa cuyo nivel de preferencia sea mayor.

Formulando el problema.**

Entonces, dado dos conjuntos, donde uno adoptara el papel de **solicitantes**, mientras que el otro tomara la identidad de los **requeridos**, se desea establecer un conjunto de relaciones uno a uno, sin repeticiones, entre solicitantes y requeridos, de acuerdo a su grado de preferencia y donde no se encuentren lo que Gale y Shapley definieron como inestabilidad (pares de parejas disconformes, con preferencias por la pareja del opuesto).

Algoritmo.

```

Comienza con 0 parejas armadas.
Mientras exista solicitantes libres y que no hayan preguntado a todos los requeridos.
    Un solicitante s que aun no tiene pareja convoca al requerido de mayor preferencia.
        Si el requerido no tiene pareja, acepta.
        Si tiene pareja, pero prefiere al solicitante s por encima del actual, rompe el vinculo, y forma pareja con el solicitante.
    El desplazado regresa al conjunto de solitantes sin pareja.

```

Consecuencias claves del algoritmo.**

1. Una vez que un requerido esta en pareja, jamas volverá a estar solo.
2. Las parejas de los solicitantes tienden a empeorar con el tiempo.
3. El algoritmo termina en a lo sumo n^2 pasos. Esto es así dado que en el peor de los casos, cada solicitante debe recorrer toda su lista de requeridos en busca de una pareja estable. Si ambos grupos tienen n unidades, entonces se obtiene que se realizaran un total de n^2 consultas.
4. El resultado es un matching perfecto. Es decir, todos tienen pareja. Para demostrar esto, vamos a suponer que existe un caso en el que un solicitante no tiene pareja y ya preguntó a todos sus requeridos. Por el punto 1, podemos inferir que si nadie quiso ser pareja de nuestro solitario solicitante, es porque todos los requeridos ya tienen pareja. Pero eso implicaría que existen n solicitantes en pareja mas uno sin pareja. Esto es absurdo porque partimos de la idea de que ambos grupos tienen n individuos. Entonces no existe un caso en que quede un solicitante sin pareja al final del algoritmo y esto significa tener un matching perfecto.
5. El matching final no tiene inestabilidades, es decir, no existen pares de parejas disconformes que prefieran al compañero del otro por encima del actual. Para demostrar esto supondremos que existe un caso en el cual el algoritmo finalizo con una inestabilidad del tipo $(s, r), (s', r') \in M$ donde M es el conjunto de parejas. y donde s y r' se prefieren antes que a sus actuales parejas. Pero si esto es así, s tuvo que convocar a r' antes y ser rechazado. Si no lo hizo, no siguió el algoritmo, y si lo hizo, entonces r' debió rechazarlo por alguien mejor y no por alguien peor. Esto es absurdo y por tanto, este caso no existe.
6. El matching es estable. Es decir, es perfecto y no posee inestabilidades.

Antes de continuar, necesitamos aclarar algunos conceptos.

- Un mismo problema puede tener varios matching estables.
 - Decimos que r es un compañero valido de s (valid partner) si existe un matching estable M que contenga a la pareja (s, r) .
 - Decimos que r es el mejor compañero de s (best partner) si s prefiere a r por encima de todos sus compañeros validos (posibles parejas en otros matching estables)
 - Análogamente, se puede plantear lo inverso.
7. El algoritmo siempre retorna un matching con los mejores compañeros para los solicitantes. Para demostrar esto, supondremos que existe un caso en el que existe un matching estable M donde s no quedó emparejado con su mejor compañero. Es decir, quedó emparejado con otro compañero valido. Entonces en M quedó una pareja del tipo (s', r) donde r prefiere a s' por encima de s . Dado que s y r son parejas validas, existe un matching estable M' compuesto por las parejas (s, r) y (s', r') . Cuando se genera M , necesariamente s' tuvo que convocar primero a r antes que a r' (ya que ambas son parejas validas, y se quedan con el que primero se encuentran) por tanto s' prefiere a r por encima de

r' , pero esto es lo que define una inestabilidad en M' , y esto es absurdo porque es estable, por tanto, este caso no existe.

8. Partiendo de 7, se puede probar que los requeridos terminan siempre terminan con su peor pareja.

Conclusiones.

En este pequeño articulo, pudimos generar el pseudocódigo de un algoritmo, obtuvimos su orden de ejecución (cantidad de pasos que requiere su ejecución con una entrada de tamaño N), sus consecuencias y la demostración de cada una. Esto es una pequeña muestra de lo que se obtiene con un algoritmo y su análisis a un nivel profundo.

Bases del análisis de algoritmos.



El foco de estos artículos es encontrar algoritmos eficientes para problemas computaciones.

Consideramos eficiente un algoritmo si este se ejecuta rápidamente con una serie de datos reales.

Este seria nuestro primer enfoque, pero tiene algunas debilidades: que significa que corra rapido? tiene que hacerlo en cualquier maquina? tenemos que considerar una maquina en particular? y que pasa con los datos con los que trabaja, deben ser muchos o pocos?

El tiempo de ejecución en el peor de los casos.

Para obtener una mejor definición de que significa que un algoritmo sea eficiente, vamos a centrarnos en como escala el mismo con N datos de entrada (o con una entrada de tamaño N). Por dar un ejemplo, ¿cuanto debería tardar una búsqueda con N datos?.

Ahora, ¿que pasa cuando N es un numero bastante grande? ¿como se comporta el algoritmo cuando todo lo que puede salir mal, sale mal? Este sera un poco nuestro enfoque para demostrar que tan eficiente es un algoritmo.

Siguiendo con el ejemplo de la búsqueda (búsqueda lineal o por fuerza bruta, es decir, pasamos por cada elemento y nos preguntamos si es o no lo que buscamos), si el elemento que buscamos esta al final de nuestra lista, entonces ejecutaríamos N pasos hasta encontrar el mismo en el peor de los casos.

Esto quiere decir que cuanto mas grande sea N , mas tardara nuestro algoritmo en encontrar el elemento. Es decir, escala de manera proporcional.

Nota: en cierta forma, el hecho de que podamos analizar los algoritmos por como se comportar frente a distintos tamaños de entrada, nos habla un poco acerca de la naturaleza discreta de los problemas computacionales.

El tiempo polinomial como una defunciones de eficiencia.

Dada una entrada de tamaño N , cuantos "pasos" debería dar nuestro algoritmo hasta terminar?. Esperamos que no sea el doble de pasos o el triple, pero depende de cada algoritmo y su peor caso. En cierta forma esperamos que sea proporcional a un factor de C , pero se sabe que puede llegar a tardar un proporcional de N^d como es el caso del algoritmo de Gale-Shapley (hay un artículo que habla sobre este caso) que ejecuta N^2 pasos en el peor de los casos.

Aritméticamente hablando, podemos formular lo siguiente:

Un algoritmo es eficiente si este tiene un tiempo polinomial de ejecución en el peor de los casos.

COMPLEJIDAD

$\log(N)$

N

$N \log(N)$

N^2

N^3

2^N

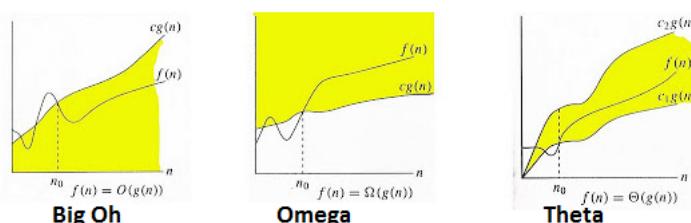
$N!$

En la tabla quedan especificados una serie de funciones comunes en análisis de algoritmos dispuestos desde el mejor hasta el peor de los casos en términos de escalabilidad y se conocen como "ordenes de ejecución".

Complejidad computacional.

Evaluaremos la eficiencia de un algoritmo de acuerdo a una función cuyo parámetro es el tamaño de la entrada $g(n)$. Nos interesa conocer el crecimiento de esa función a medida que n crece. Llamaremos a la misma Orden asintótico de crecimiento. Podemos acotar ese crecimiento utilizando otras funciones conocidas. Cotas: O , Θ , Ω . (conocidos como big o, big theta, big omega).

- **Límite superior:** para la cota superior diremos que, sea una función $f(n)$ que representa la eficiencia del algoritmo y $g(n)$ una función de caracterización. Diremos que $f(n)$ es $O(g(n))$, si y solo si para $n_0 \geq 0$, $C > 0$ tal que para todo $n \geq n_0 : f(n) \leq C \cdot g(n)$.
- **Límite inferior:** para la cota inferior diremos que, sea una función $f(n)$ que representa la eficiencia del algoritmo y $g(n)$ una función de caracterización. Diremos que $f(n)$ es $\Omega(g(n))$, si y solo si, para $n_0 \geq 0$, $C > 0$ tal que para todo $n \geq n_0 : f(n) \geq C \cdot g(n)$.
- **Promedio:** para el promedio diremos que, sea una función $f(n)$ que representa la eficiencia del algoritmo y $g(n)$ una función de caracterización. Diremos que $f(n)$ es $\Theta(g(n))$, si y solo si, para $n_0 \geq 0$, $c_1 > 0$, $c_2 > 0$ tal que para todo $n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.



Entonces existen tres ordenes de ejecución que acompañan a cada algoritmo, el que señala el mejor de los casos, el del peor de los casos y el caso medio. Generalmente, siempre se usa la notación O para señalar el peor de los casos que es el que nos interesa. Entonces un algoritmo puede ser de orden:

ORDENES DE EJECUCIÓN.

$O(\log(N))$

$O(N)$

$O(N\log(N))$

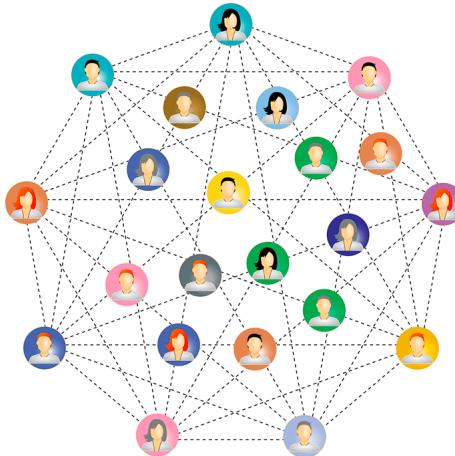
$O(N^2)$

$O(N^3)$

$O(2^N)$

$O(N!)$

Teoría de grafos.



En este artículo se dará un breve repaso de los conceptos más importantes sobre grafos, pero se dará más importancia a los problemas que nos pueden presentar y las aplicaciones para los cuales fueron estudiados.

Definiciones básicas.

Un grafo G es simplemente una forma de codificar relaciones por pares entre un conjunto de objetos: consiste en una colección V de nodos y una colección E de aristas o aristas, cada uno de los cuales "une" dos de los nodos. Por lo tanto, representamos una arista $e \in E$ como un subconjunto de dos elementos de V : $e = \{u, v\}$ para algunos $u, v \in V$, donde llamamos u y v los extremos de e .

Las aristas en un grafo indican una relación simétrica entre sus extremos. A menudo queremos codificar relaciones asimétricas, y para esto utilizamos la noción estrechamente relacionada de un grafo dirigido. Un grafo dirigido G' consiste en un conjunto de nodos V y un conjunto de aristas dirigidas E' . Cada $e' \in E'$ es un par ordenado (u, v) , en otras palabras, los roles de u y v no son intercambiables, y llamamos u la cola de la arista y v la cabeza. También diremos que la arista e' sale del nodo u y entra en el nodo v .

Cuando queremos enfatizar que el grafo que estamos considerando no está dirigido, lo llamaremos un grafo no dirigido; por defecto, sin embargo, el término "grafo" significará un gráfico no dirigido. También vale la pena mencionar dos advertencias en nuestro uso de la terminología grafo. Primero, aunque una arista e en un grafo no dirigido debe escribirse correctamente como un conjunto de nodos $\{u, v\}$, uno lo verá escrito con más frecuencia en la notación utilizada para pares ordenados: $e = (u, v)$. Segundo, un nodo en un grafo también se llama frecuentemente vértice; En este contexto, las dos palabras tienen exactamente el mismo significado.

Caminos y conectividad.

Una de las operaciones fundamentales en un grafo es atravesar una secuencia de nodos conectados por aristas. Definimos una ruta o camino en un grafo no dirigido $G = (V, E)$ como una secuencia P de nodos $v_1, v_2, \dots, v_{k-1}, v_k$ con la propiedad de que cada par consecutivo v_i, v_{i+1} está unido por una arista en G . A menudo se denomina ruta de v_1 a v_k . Una ruta se llama simple si todos sus vértices son distintos entre sí. Un ciclo es un camino $v_1, v_2, \dots, v_{k-1}, v_k$ en el que $k > 2$, los primeros $k - 1$ nodos son todos distintos, y $v_1 = v_k$, en otras palabras, la secuencia de nodos "regresa" a donde comenzó. Todas estas definiciones se trasladan naturalmente a gráficos dirigidos, con el siguiente cambio: en una ruta o ciclo dirigido, cada par de nodos consecutivos tiene la propiedad de que (v_i, v_{i+1}) es una arista. En otras palabras, la secuencia de nodos en la ruta o ciclo debe respetar la direccionalidad de los aristas.

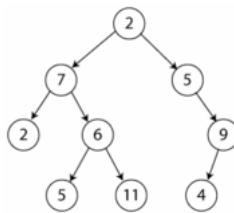
Decimos que un grafo no dirigido está conectado si, para cada par de nodos u y v , hay una ruta de u a v . Elegir cómo definir la conectividad de un grafo dirigido es un poco más sutil, ya que es posible que usted tenga una ruta hacia v mientras que v no tiene ruta hacia u . Decimos que un gráfico dirigido está fuertemente conectado si, por cada dos nodos u y v , hay una ruta de u a v y una ruta de v a u .

Además de simplemente conocer la existencia de una ruta entre un par de nodos u y v , también podemos querer saber si hay una ruta corta. Por lo tanto, definimos la distancia entre dos nodos u y v para que sea el número mínimo de aristas en una ruta u - v . (Podemos designar algún símbolo como ∞ para denotar la distancia entre nodos que no están conectados por una ruta). El término distancia aquí proviene de imaginar que G representa una red de comunicación o transporte; si queremos llegar de u a v , es posible que queramos una ruta con la menor cantidad posible de "saltos".

Arboles.

Decimos que un gráfico no dirigido es un árbol si está conectado y no contiene un ciclo. Por ejemplo, los dos gráficos representados en la Figura 3.1 son árboles. En un sentido fuerte, los árboles son el tipo más simple de gráfico conectado: eliminar cualquier arista de un árbol lo desconectará.

Para pensar en la estructura de un árbol T , es útil enraizarlo en un nodo particular r . Físicamente, esta es la operación de agarrar T en el nodo r y dejar que el resto cuelgue hacia abajo bajo la fuerza de la gravedad, como un móvil. Más precisamente, "orientamos" cada arista de T lejos de r ; para cada otro nodo v , declaramos que el padre de v es el nodo u que precede directamente a v en su ruta desde r ; declaramos que w es hijo de v si v es el padre de w . En términos más generales, decimos que w es un descendiente de v (o v es un antepasado de w) si v se encuentra en el camino de la raíz a w ; y decimos que un nodo x es una hoja si no tiene descendientes.



A continuación damos algunas propiedades básicas y obvias de todo árbol.

Todo árbol con n nodos, tiene exactamente $n - 1$ aristas.

De hecho, la siguiente afirmación más fuerte es cierta, aunque no la probamos aquí.

Dado un grafo no dirigido G con n nodos, con dos de las siguientes declaraciones implica la tercera.

1. G está conectado.
2. G no tiene ciclos.
3. G tiene $n - 1$ aristas.

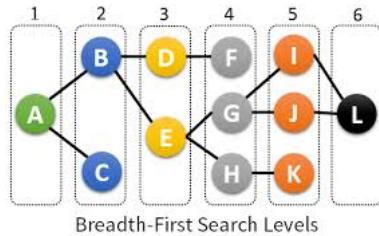
Conejividad en un grafo y grafo transversal.

Habiendo desarrollado algunas nociones fundamentales con respecto a los gráficos, pasamos a una pregunta algorítmica muy básica: conectividad de nodo a nodo. Supongamos que se nos da un gráfico $G = (V, E)$ y dos nodos particulares s y t . Nos gustaría encontrar un algoritmo eficiente que responda a la pregunta: ¿Hay una ruta de s a t en G ? Llamaremos a esto el problema de determinar la conectividad s - t .

En esta sección, describimos dos algoritmos naturales para este problema en un nivel alto: búsqueda de amplitud primero (BFS) y búsqueda de profundidad primero (DFS). En la siguiente sección, discutimos cómo implementar cada uno de estos de manera eficiente, construyendo sobre una estructura de datos para representar un gráfico como la entrada a un algoritmo.

Búsqueda de amplitud primero.

Quizás el algoritmo más simple para determinar la conectividad s - t es la búsqueda por amplitud (BFS), en la que exploramos hacia afuera desde s en todas las direcciones posibles, agregando nodos una "capa" a la vez. Por lo tanto, comenzamos con s e incluimos todos los nodos unidos por una arista a s ; esta es la primera capa de la búsqueda. Luego incluimos todos los nodos adicionales que están unidos por una arista a cualquier nodo en la primera capa; esta es la segunda capa. Continuamos de esta manera hasta que no se encuentren nuevos nodos.



Podemos definir las capas L_1, L_2, L_3, \dots construidas por el algoritmo BFS con mayor precisión de la siguiente manera.

- La capa L_1 consta de todos los nodos que son vecinos de s . (Por razones de notación, a veces usaremos la capa L_0 para denotar el conjunto que consiste solo en s).
- Suponiendo que hemos definido las capas L_1, \dots, L_j , la capa L_{j+1} consta de todos los nodos que no pertenecen a una capa anterior y que tienen una arista a un nodo en la capa L_j .

Recordando nuestra definición de la distancia entre dos nodos como el número mínimo de aristas en una ruta que los une, vemos que la capa L_1 es el conjunto de todos los nodos a la distancia 1 de s , y más generalmente la capa L_j es el conjunto de todos los nodos en distancia exactamente j de s . Un nodo no aparece en ninguna de las capas si y solo si no hay una ruta hacia él. Por lo tanto, BFS no solo determina los nodos que pueden alcanzar, sino que también calcula las rutas más cortas para llegar a ellos. Resumimos esto en el siguiente hecho.

Para cada $j \geq 1$, la capa L_j producida por BFS consiste en todos los nodos a una distancia exactamente j de s . Hay una ruta de s a t si y solo si t aparece en alguna capa.

Otra propiedad de la búsqueda de amplitud es que produce, de una manera muy natural, un árbol T enraizado en s en el conjunto de nodos accesibles desde s . Llamamos al árbol T que se produce de esta manera un árbol de búsqueda de amplitud.

Sea T un árbol de búsqueda de amplitud, que x e y sean nodos en T que pertenezcan a las capas L_i y L_j respectivamente, y que (x, y) sea una arista de G . Entonces i y j difieren en a lo sumo 1.

Demostración: Supongamos, por contradicción, que i y j difieren en más de 1; en particular, suponga que $i < j - 1$. Ahora considere el punto en el algoritmo BFS cuando se examinaron los aristas incidentes con x . Como x pertenece a la capa L_i , los únicos nodos descubiertos a partir de x pertenecen a las capas $L_i + 1$ y anteriores; por lo tanto, si y es un vecino de x , entonces debería haberse descubierto en este punto a más tardar y , por lo tanto, debería pertenecer a la capa $L_i + 1$ o anterior.

Búsqueda de profundidad primero.

Otro método natural para encontrar los nodos accesibles desde s es el enfoque que podría tomar si el gráfico G fuera realmente un laberinto de habitaciones interconectadas y estuviera caminando por él. Comenzaría desde s y probaría el primer arista que sale de él, a un nodo v . Luego, seguiría el primer arista que sale de v , y continuaría de esta manera hasta llegar a un "callejón sin salida": un nodo para lo cual ya habías explorado a todos sus vecinos. Luego retrocedería hasta llegar a un nodo con un vecino inexplorado, y reanudaría desde allí. A este algoritmo lo llamamos búsqueda de profundidad primero (DFS), ya que explora G yendo lo más profundo posible y solo retrocediendo cuando es necesario.

DFS también es una implementación particular del algoritmo genérico de crecimiento de componentes que presentamos anteriormente. Se describe más fácilmente en forma recursiva: podemos invocar DFS desde cualquier punto de partida pero mantener el conocimiento global de qué nodos ya se han explorado.

```
DFS(u):
    Marcar u como explorado y agregar u a R.
    Para cada arista e = (u, v) incidentes a u:
        si $v$ no fue marcado como explorado:
            invocar recursivamente a DFS(v).
```

Llamamos R a los componentes conexos en G (o de T , o de cualquier grafo que se encuentre en plena formación), estos se pueden formar por varios algoritmos, entre ellos el BFS y el DFS.

Para aplicar esto a la conectividad $s-t$, simplemente declaramos que todos los nodos inicialmente no serán explorados e invocamos DFS (s).

Hay algunas similitudes fundamentales y algunas diferencias fundamentales entre DFS y BFS. Las similitudes se basan en el hecho de que ambas construyen el componente conectado que contiene s , y veremos en la siguiente sección que logran niveles de eficiencia cualitativamente similares.

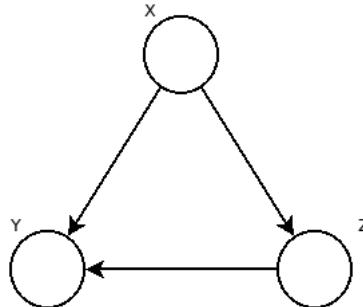
Si bien DFS finalmente visita exactamente el mismo conjunto de nodos que BFS, generalmente lo hace en un orden muy diferente; explora su camino por caminos largos, potencialmente alejándose mucho de s , antes de retroceder para intentar acercarse a nodos más inexplorados. Podemos ver un reflejo de esta diferencia en el hecho de que, al igual que BFS, el algoritmo DFS produce un árbol T enraizado natural en el componente que contiene s , pero el árbol generalmente tendrá una estructura muy diferente. Hacemos s la raíz del árbol T , y hacemos a u padre de v cuando u es responsable del descubrimiento de v . El árbol resultante se denomina árbol de búsqueda de profundidad primero del componente R .

Propiedad fundamental de DFS: Para una llamada recursiva $DFS(u)$ dada, todos los nodos que están marcados como "Explorados" entre la invocación y el final de esta llamada recursiva son descendientes de u en T .

Otra propiedad de los árboles DFS es la siguiente:

Sea un árbol T de DFS formado a partir de el grafo G . Y sean x e y dos nodos tales que existe una arista (x, y) en G , pero no en T , se puede decir que x o y son ancestros uno del otro.

Demostración: Este es una demostración que puede ser mejor entendida con un gráfico. Suponga que (x, y) es una arista de G que no es una arista de T , y suponga sin pérdida de generalidad que x es alcanzado primero por el algoritmo DFS. Cuando se examina la arista (x, y) durante la ejecución de $DFS(x)$, no se agrega a T porque y está marcado como "Explorado". Como supusimos que x fue alcanzado primero por DFS, se puede decir que y es un nodo que se descubrió entre la invocación y el final de la llamada recursiva $DFS(x)$ anterior. De la propiedad anterior se deduce que y es un descendiente de x .



Para exemplificar, si se ejecuta DFS sobre el diagrama propuesto, arrancando por x y al alcanzar la arista (x,y) , y fue encontrada como "explorada", no queda otra cosa para decir mas que y ya fue encontrada por otro camino en el proceso DFS (por ejemplo, el camino $x-z-y$). Entonces x es un ancestro de y .

Conejividad en grafos dirigidos.

Hasta ahora, hemos estado analizando problemas en gráficos no dirigidos; Ahora consideraremos hasta qué punto estas ideas se trasladan al caso de los gráficos dirigidos.

Algoritmos de búsqueda en grafos.

La búsqueda por amplitud y la búsqueda por profundidad son casi las mismas en gráficos dirigidos que en gráficos no dirigidos. Nos centraremos aquí en BFS. Comenzamos en un nodo s , definimos una primera capa de nodos que constará de todos aquellos a los que s tiene una arista, definimos una segunda capa que constará de todos los nodos adicionales a los que estos nodos de la primera capa tienen una arista, y así sucesivamente. De esta manera, descubrimos nodos capa por capa a medida que se alcanzan en esta búsqueda externa desde s , y los nodos en la capa j son precisamente aquellos para los cuales el camino más corto desde s tiene exactamente j aristas. Como en el caso no dirigido, este algoritmo realiza un trabajo constante para cada nodo y arista, lo que resulta en un tiempo de ejecución de $O(m + n)$.

Es importante comprender qué está computando esta versión dirigida de BFS. En los gráficos dirigidos, es posible que un nodo s tenga una ruta hacia un nodo t aunque t no tenga una ruta hacia s ; y lo que BFS dirigido está computando es el conjunto de todos los nodos t con la propiedad que s tiene una ruta hacia t . Dichos nodos pueden o no tener rutas de regreso a s .

También existe un análogo natural de búsqueda en profundidad, que también se ejecuta en tiempo lineal y calcula el mismo conjunto de nodos. Es nuevamente un procedimiento recursivo que trata de explorar lo más profundamente posible, en este caso solo siguiendo los aristas de acuerdo con su dirección inherente. Por lo tanto, cuando DFS está en un nodo u , lanza recursivamente una búsqueda de profundidad primero, en orden, para cada nodo en el que tiene una arista.

Supongamos que, para un nodo dado s , deseamos el conjunto de nodos con rutas a s , en lugar del conjunto de nodos a los que s tiene rutas. Una manera fácil de hacer esto sería definir un nuevo gráfico dirigido, G^{rev} , que obtenemos de G simplemente invirtiendo la dirección de cada arista. Entonces podríamos ejecutar BFS o DFS en G^{rev} ; un nodo tiene una ruta desde s en G^{rev} si y solo si tiene una ruta hacia s en G .

Grafos fuertemente conexos.

Recuerde que un gráfico dirigido está fuertemente conectado si, por cada dos nodos u y v , hay una ruta de u a v y una ruta de v a u . También vale la pena formular alguna terminología para la propiedad en el corazón de esta definición; Digamos que dos nodos u y v en un gráfico dirigido son mutuamente accesibles si hay una ruta de u a v y también una ruta de v a u . (Por lo tanto, un gráfico está fuertemente conectado si cada par de nodos es accesible mutuamente).

La accesibilidad mutua tiene una serie de buenas propiedades, muchas de ellas derivadas del siguiente hecho simple.

Sí u y v son mutuamente accesibles, y v y w son mutuamente accesibles, entonces u y w son mutuamente accesibles.

Demostración: Para construir un camino de u a w , primero vamos de u a v (a lo largo del camino garantizado por la accesibilidad mutua de u y v), y luego de v a w (a lo largo del camino garantizado por la accesibilidad mutua de v y w). Para construir una ruta de w a u , simplemente revertimos este razonamiento: primero vamos de w a v (a lo largo del camino garantizado por la accesibilidad mutua de v y w), y luego de v a u (a lo largo del camino garantizado por la accesibilidad mutua de u y v).

Hablemos ahora de los componentes fuertemente conexos.

Para dos nodos s y t en un grafo dirigido, sus componentes fuertemente conexos son idénticos o disjuntos.

Demostración: Considere dos nodos s y t que son mutuamente accesibles; Afirmamos que los componentes fuertemente conexos que contienen s y t son idénticos. De hecho, para cualquier nodo v , si s y v son mutuamente accesibles, entonces por la propiedad fundamental de DFS, t y v también son mutuamente accesibles. De manera similar, si t y v son mutuamente accesibles, entonces nuevamente por la propiedad fundamental de DFS, s y v son mutuamente accesibles.

Por otro lado, si s y t no son accesibles mutuamente, entonces no puede haber un nodo v que esté en el componente fuertemente conexo de cada uno. Porque si existiera tal nodo v , entonces s y v serían mutuamente accesibles, y v y t serían mutuamente accesibles, por lo que a partir de la propiedad fundamental de DFS se deduciría que s y t serían mutuamente accesibles.

De hecho, aunque no discutiremos los detalles de esto aquí, con más trabajo es posible calcular los componentes fuertemente conexos para todos los nodos en un tiempo total de $O(m + n)$.

Grafos dirigidos acíclicos y el orden topológico.

Si un grafo no dirigido no tiene ciclos, entonces tiene una estructura extremadamente simple: cada uno de sus componentes conectados es un árbol. Pero es posible que un grafo dirigido no tenga ciclos (dirigidos) y aún tenga una estructura muy rica. Por ejemplo, tales gráficos pueden tener una gran cantidad de aristas: si comenzamos con el conjunto de nodos $\{1, 2, \dots, n\}$ e incluimos una arista (i, j) siempre que $i < j$, entonces el grafo dirigido resultante tiene $\binom{n}{2}$ aristas. Pero no hay ciclos.

El problema del orden topológico.

Los DAG son una estructura muy común en informática, porque muchos tipos de redes de dependencia del tipo que discutimos son acíclicas. Por lo tanto, los DAG se pueden usar para codificar relaciones de precedencia o dependencias de forma natural. Supongamos que tenemos un conjunto de tareas etiquetadas $\{1, 2, \dots, n\}$ que deben realizarse, y hay dependencias entre ellas que estipulan, para ciertos pares i y j , que debo realizar antes de j . Por ejemplo, las tareas pueden ser cursos, con requisitos previos que establecen que ciertos cursos deben tomarse antes que otros. O las tareas pueden corresponder a una tubería de trabajos informáticos, con afirmaciones de que la salida del trabajo i se usa para determinar la entrada al trabajo j , y por lo tanto, el trabajo que debo hacer antes del trabajo j .

Podemos representar un conjunto de tareas tan interdependientes mediante la introducción de un nodo para cada tarea y una arista dirigido (i, j) siempre que deba realizarse antes de j . Si la relación de precedencia es significativa, el gráfico resultante G debe ser un DAG. De hecho, si contuviera un ciclo C , no habría forma de realizar ninguna de las tareas en C : dado que cada tarea en C no puede comenzar hasta que se complete otra, no se puede realizar ninguna tarea en C , ya que ninguna se puede hacer primero.

Continuemos un poco más con esta imagen de los DAG como relaciones de precedencia. Dado un conjunto de tareas con dependencias, sería natural buscar un orden válido en el que se pudieran realizar las tareas, de modo que se respeten todas las dependencias. Específicamente, para un gráfico dirigido G , decimos que un ordenamiento topológico de G es un ordenamiento de sus nodos como v_1, v_2, \dots, v_n , de modo que para cada arista (v_i, v_j) , tenemos $i < j$. En otras palabras, todas las aristas apuntan "hacia adelante" en el orden. Un orden topológico en las tareas proporciona un orden en el que se pueden realizar de forma segura; cuando llegamos a la tarea v_j , todas las tareas que se requieren para precederla ya se han realizado.

De hecho, podemos ver un orden topológico de G como una "prueba" inmediata de que G no tiene ciclos, a través de lo siguiente.

Principal propiedad del orden topológico: Si G tiene un orden topológico, entonces G es un DAG.

Demostración: Supongamos, por contradicción, que G tiene un orden topológico v_1, v_2, \dots, v_n , y también tiene un ciclo C . Sea v_i el nodo de índice más bajo en C , y sea v_j el nodo en C justo antes de v_i . Esto significa que (v_j, v_i) es una arista. Pero al elegir i , tenemos $j > i$, lo que contradice la suposición de que v_1, v_2, \dots, v_n era un ordenamiento topológico.

Partiendo de la propiedad anterior, la pregunta principal que consideramos aquí es su inversa ¿Cada DAG tiene un orden topológico y, de ser así, cómo podemos encontrar uno de manera eficiente? Un método para hacer esto para cada DAG sería muy útil: mostraría que para cualquier relación de precedencia en un conjunto de tareas sin ciclos, existe un orden computable eficiente en el que realizar las tareas.

Diseño y análisis del algoritmo.

De hecho, lo contrario de la propiedad de todo orden topológico se cumple, y lo establecemos mediante un algoritmo eficiente para calcular un orden topológico. La clave de esto radica en encontrar una manera de comenzar: ¿qué nodo colocamos al comienzo del ordenamiento topológico? Tal nodo v_1 necesitaría no tener aristas entrantes, ya que cualquier arista entrante violaría la propiedad definitoria del ordenamiento topológico, que todas las aristas apuntan hacia adelante. Por lo tanto, tenemos que demostrar el siguiente hecho.

En cada DAG G , hay un nodo v sin aristas entrantes.

Demostración: Sea G un gráfico dirigido en el que cada nodo tiene al menos un arista entrante. Mostramos cómo encontrar un ciclo en G ; Esto demostrará el reclamo. Seleccionamos cualquier nodo v , y comenzamos a seguir las aristas hacia atrás desde v : dado que v tiene al menos un arista entrante (u, v) , podemos caminar hacia atrás hacia u ; entonces, dado que u tiene al menos una arista entrante (x, u) , podemos caminar hacia atrás a x ; y así. Podemos continuar este proceso indefinidamente, ya que cada nodo que encontramos tiene una arista entrante. Pero después de $n + 1$ pasos, habremos visitado algún nodo w dos veces. Si dejamos que C denote la secuencia de nodos encontrados entre visitas sucesivas a w , entonces claramente C forma un ciclo.

De hecho, la existencia de tal nodo v es todo lo que necesitamos para producir un ordenamiento topológico de G por inducción. Específicamente, afirmemos por inducción que cada DAG tiene un orden topológico. Esto es claramente cierto para los DAG en uno o dos nodos. Ahora suponga que es cierto para DAG con hasta cierto número de nodos n . Luego, dado un DAG G en $n + 1$ nodos, encontramos un nodo v sin aristas entrantes, como lo garantiza la propiedad anterior. Colocamos v primero en el orden topológico; esto es seguro, ya que todas las aristas de v apuntarán hacia adelante. Ahora $G - \{v\}$ es un DAG, ya que eliminar v no puede crear ningún ciclo que no estuviera allí anteriormente.

Además, $G - \{v\}$ tiene n nodos, por lo que podemos aplicar la hipótesis de inducción para obtener un ordenamiento topológico de $G - \{v\}$. Anexamos los nodos de $G - \{v\}$ en este orden después de v ; Este es un ordenamiento de G en el que todas las aristas apuntan hacia adelante y, por lo tanto, es un ordenamiento topológico.

Si G es un DAG, entonces G tiene un orden topológico.

Demostración: La prueba de esta propiedad es inductiva y se percibe en el siguiente algoritmo que computa el orden topológico de cualquier grafo G .

```
Con una lista L inicialmente vacía.  
OrdenTopologico(G, L):  
    Encontrar el nodo $v$ que no posea aristas entrantes.  
    Eliminar $v$ de G.  
    Agregar $v$ a L.  
    OrdenTopologico(G-{v}, L).
```

Para limitar el tiempo de ejecución de este algoritmo, observamos que identificar un nodo v sin aristas entrantes y eliminarlo de G se puede hacer en tiempo $O(n)$. Como el algoritmo se ejecuta durante n iteraciones, el tiempo total de ejecución es $O(n^2)$.

Este no es un mal tiempo de ejecución; y si G es muy denso y contiene aristas $\Theta(n^2)$, entonces es lineal en el tamaño de la entrada. Pero bien podríamos querer algo mejor cuando el número de aristas m es mucho menor que n^2 . En tal caso, un tiempo de ejecución de $O(m + n)$ podría ser una mejora significativa sobre $\Theta(n^2)$.

De hecho, podemos lograr un tiempo de ejecución de $O(m + n)$ utilizando el mismo algoritmo de alto nivel, eliminando iterativamente nodos sin aristas entrantes.

Simplemente tenemos que ser más eficientes para encontrar estos nodos, y hacemos esto de la siguiente manera.

Declaramos que un nodo está "activo" si aún no ha sido eliminado por el algoritmo, y mantenemos explícitamente dos cosas:

1. Para cada nodo w , el número de aristas entrantes que w tiene de los nodos activos; y
2. El conjunto S de todos los nodos activos en G que no tienen aristas entrantes de otros nodos activos.

Al principio, todos los nodos están activos, por lo que podemos inicializar (1) y (2) con una sola pasada a través de los nodos y las aristas. Luego, cada iteración consiste en seleccionar un nodo v del conjunto S y eliminarlo.

Después de eliminar v , pasamos por todos los nodos w a los cuales v tenía una arista, y restamos uno del número de aristas entrantes activos que mantenemos para w . Si esto hace que el número de aristas entrantes activas w caiga a cero, entonces agregamos w al conjunto S . Continuando de esta manera, hacemos un seguimiento de los nodos que son elegibles para eliminación en todo momento, mientras dedicamos un trabajo constante por arista. en el transcurso de todo el algoritmo.

Prueba de bipartidismo: una aplicación de búsqueda en amplitud.

Recordemos la definición de un gráfico bipartito: es uno en el que el conjunto de nodos V se puede dividir en conjuntos X e Y de tal manera que cada arista tiene un extremo en X y el otro extremo en Y . Para hacer la discusión un poco más suave, podemos imaginar que los nodos en el conjunto X son de color rojo, y los nodos en el conjunto Y son de color azul. Con estas imágenes, podemos decir que un gráfico es bipartito si es posible colorear sus nodos de rojo y azul para que cada arista tenga un extremo rojo y uno azul.

El problema del grafo bipartito.

En los capítulos anteriores, vimos ejemplos de gráficos bipartitos. Aquí comenzamos preguntando: ¿Cuáles son algunos ejemplos naturales de un gráfico no bipartito, uno donde no es posible tal partición de V ?

Claramente, un triángulo no es bipartito, ya que podemos colorear un nodo rojo, otro azul, y luego no podemos hacer nada con el tercer nodo. En términos más generales, considere un ciclo C de longitud impar, con nodos numerados 1, 2, 3, ..., $2k$, $2k + 1$. Si coloreamos el nodo 1 en rojo, entonces debemos colorear el nodo 2 en azul, y luego debemos colorear el nodo 3 rojo, y así sucesivamente: colorear los nodos impares rojos y los nodos pares azules. Pero luego debemos colorear el nodo $2k + 1$ rojo, y tiene una arista al nodo 1, que también es rojo. Esto demuestra que no hay forma de dividir C en nodos rojos y azules según sea necesario. En términos más generales, si un gráfico G simplemente contiene un ciclo impar, entonces podemos aplicar el mismo argumento; así hemos establecido lo siguiente.

Si un grafo G es bipartito, entonces este no puede contener un ciclo de longitud par.

Es fácil reconocer que un gráfico es bipartito cuando los conjuntos apropiados X e Y (es decir, nodos rojos y azules) realmente se han identificado para nosotros; y en muchos entornos donde surgen gráficos bipartitos, esto es natural. Pero supongamos que encontramos un gráfico G sin ninguna anotación provista por nosotros, y nos gustaría determinar por nosotros mismos si es bipartito, es decir, si existe una partición en los nodos rojo y azul, según sea necesario. ¿Qué tan difícil es esto? Un ciclo impar es un simple "obstáculo" para que un gráfico sea bipartito. ¿Existen otros obstáculos más complejos para la bipartididad?

Diseño del algoritmo.

De hecho, existe un procedimiento muy simple para evaluar la bipartididad, y su análisis puede usarse para mostrar que los ciclos impares son el único obstáculo. Primero suponemos que el gráfico G está conectado, ya que de lo contrario podemos primero calcular sus componentes conectados y analizar cada uno de ellos por separado. A continuación, seleccionamos cualquier nodo $s \in V$ y lo coloreamos de rojo; no hay pérdida al hacer esto, ya que s debe recibir algo de color. Se deduce que todos los vecinos de s deben ser de color azul, así que hacemos esto. Luego se deduce que todos los vecinos de estos nodos deben ser de color rojo, sus vecinos deben ser de color azul, y así sucesivamente, hasta que se coloree todo el gráfico. En este punto, tenemos un color rojo / azul válido de G , en el que cada arista tiene extremos de colores opuestos, o hay algún arista con extremos del mismo color. En este último caso, parece claro que no hay nada que pudieramos haber hecho: G simplemente no es bipartito. Ahora queremos argumentar este punto con precisión y también encontrar una manera eficiente de realizar la coloración.

Lo primero que debe notar es que el procedimiento de coloración que acabamos de describir es esencialmente idéntico a la descripción de BFS: nos movemos hacia afuera desde s , coloreando nodos tan pronto como los encontramos por primera vez. De hecho, otra forma de describir el algoritmo de coloración es la siguiente: realizamos BFS, coloreamos rojo, toda la capa L1 azul, toda la capa L2 roja, y así sucesivamente, coloreamos capas impares azules y capas pares rojas.

Podemos implementar esto sobre BFS, simplemente tomando la implementación de BFS y agregando una matriz de color adicional sobre los nodos. Cada vez que llegamos a un paso en BFS donde estamos agregando un nodo v a una lista $L[i+1]$, asignamos $\text{Color}[v] = \text{rojo}$ si $i+1$ es un número par, y $\text{Color}[v] = \text{azul}$ si $i+1$ es un número impar. Al final de este procedimiento, simplemente escaneamos todos las aristas y determinamos si hay algún arista para el que ambos extremos recibieron el mismo color. Por lo tanto, el tiempo total de ejecución del algoritmo de coloración es $O(m+n)$, tal como lo es para BFS.

Análisis del algoritmo.

Sea G un gráfico conectado, y sea L_1, L_2, \dots las capas producidas por BFS comenzando en el nodo s . Entonces exactamente una de las siguientes dos cosas debe cumplir.

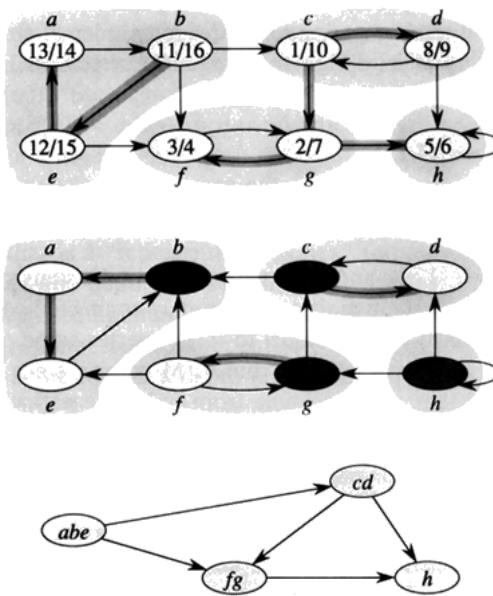
- No existen en G aristas que unan dos nodos de distintas capas. En este caso, las capas pares pueden ser coloreadas de una forma, y las capas impares pueden ser coloreadas de otra forma.

- Si existe una arista en G que une dos nodos de la misma capa, entonces G contiene un ciclo de largo impar y por tanto no puede ser bipartito.

Algoritmo de Kosaraju para componentes fuertemente conexas.

Ahora consideraremos una aplicación clásica de búsqueda en profundidad: descomponer un gráfico dirigido en sus componentes fuertemente conectados. Esta sección muestra cómo hacerlo mediante dos búsquedas en profundidad. Muchos algoritmos que funcionan con grafos dirigidos comienzan con tal descomposición. Después de descomponer el grafo en componentes fuertemente conexos, dichos algoritmos se ejecutan por separado en cada uno y luego combinan las soluciones de acuerdo con la estructura de conexiones entre los componentes.

Recordemos que un componente fuertemente conexo de un grafo dirigido $G = (V, E)$ es un máximo conjunto de vértices $C \subseteq V$ tal que por cada par de vértices u y v en C , se tiene un camino que une u con v y v con u . Es decir, ambos son accesibles entre si.



En las figuras anteriores (que las llamaremos como a , b y c según su orden) se muestra un grafo dirigido G . En la figura (a) se muestra al grafo dirigido G inicial. Cada región sombreada es un componente fuertemente conexo de G . Cada vértice está etiquetado con sus tiempos de descubrimiento y acabado en una búsqueda de profundidad (DFS), y las aristas de los árboles están sombreados.

En la figura (b) se muestra el grafo G^T , la transposición de G , con el primer bosque de profundidad calculado. Cada componente fuertemente conexo corresponde a una DFS. Los vértices b , c , g y h , que están muy sombreados, son las raíces de los primeros árboles de profundidad producidos por la búsqueda en profundidad de G^T .

En la figura (c) se muestra el grafo de componentes acíclicos o G^{SCC} obtenido al contraer todos las aristas dentro de cada componente fuertemente conexo de G de modo que solo quede un vértice en cada componente.

Nuestro algoritmo para encontrar componentes fuertemente conexos de un grafo $G = (V, E)$ usa la transposición de G , que definimos anteriormente como el grafo $G^T = (V, E^T)$, donde $E^T = \{(u, v) : (v, u) \in E\}$. Es decir, E^T consiste en las aristas de G con sus direcciones invertidas. Dada una representación de lista de adyacencias de G , el tiempo para crear G^T es $O(V + E)$. Es interesante observar que G y G^T tienen exactamente los mismos componentes fuertemente conexos: u y v son accesibles entre sí en G , si y solo si, son accesibles entre sí en G^T .

El siguiente algoritmo de tiempo lineal (es decir, $O(V + E)$) calcula los componentes fuertemente conexos de un grafo dirigido $G = (V, E)$ usando dos búsquedas de profundidad primero, una en G y otra en G^T .

```

Invocar DFS(G) para calcular los tiempos de finalización v.f de
cada vértice v.
Calcular GT.
Invocar DFS(GT), pero en el loop principal de DFS, considerar los
vértices in orden decreciente segun v.f
Imprimir los vértices de cada arbol en el bosque DFS formado en la
anterior sentencia como componentes fuertemente conexos separados.

```

La idea detrás de este algoritmo proviene de una propiedad clave de los grafos de componentes $G^{SCC} = (V^{SCC}, E^{SCC})$, que definimos de la siguiente manera. Suponga que G tiene componentes fuertemente conexos C_1, C_2, \dots, C_K . El conjunto de vértices V^{SCC} es v_1, v_2, \dots, v_k y contiene un vértice v_i para cada componente fuertemente conectado C_i de G . Hay una arista $(v_i, v_j) \in E^{SCC}$ si G contiene una arista dirigida (x, y) para algún $x \in C_i$ y algún $y \in C_j$. Visto de otra manera, al contraer todas las aristas, cuyos vértices incidentes están dentro del mismo componente fuertemente conexo de G , el gráfico resultante es G^{SCC} .

La propiedad clave es que el grafo de componentes es un grafo dirigido e implica el siguiente lema.

Sean C y C' dos componentes fuertemente conexos distintos de un grafo dirigido $G = (V, E)$, sean $u, v \in C$ y $u', v' \in C'$, y supongamos que G posee un camino que une u con u' . Entonces no puede existir en G un camino que une v' con v .

Demostración: Si G contiene un camino que une v' con v , entonces este contiene un camino que une u , u' y v' , y además, un camino que une v' , v y u . Entonces, u y v' son accesibles entre sí, pero esto contradice la suposición de que C y C' son distintos. Esto es un absurdo.

Veremos que al considerar los vértices en la segunda búsqueda de profundidad en orden decreciente según los tiempos de finalización que se calcularon en la primera búsqueda de profundidad, estamos, en esencia, visitando los vértices del grafo de componentes (cada uno de que corresponde a un componente fuertemente conectado de G) en orden topológicamente ordenado.

Debido a que el algoritmo de Kosajaru realiza dos búsquedas en profundidad, existe la posibilidad de ambigüedad cuando discutimos $u.d$ o $u.f$. En esta sección, estos valores siempre se refieren a los tiempos de descubrimiento y finalización calculados por la primera llamada de DFS.

Extendemos la notación para los tiempos de descubrimiento y finalización a conjuntos de vértices. Si $U \subseteq V$, entonces definimos $d(U) = \min_{u \in U} \{u.d\}$ y $f(U) = \max_{u \in U} \{u.f\}$. Es decir, $d(U)$ y $f(U)$ son el tiempo de descubrimiento más temprano y el último tiempo de finalización, respectivamente, de cualquier vértice en U .

El siguiente lema y su corolario otorgan una propiedad clave que relaciona componentes fuertemente conexos y tiempos de finalización en la primera búsqueda en profundidad.

Sean C_i y C_j dos componentes fuertemente conexos distintos en el grafo $G = (V, A)$. Suponga que existe una arista $(u, v) \in A$, donde $u \in C_i$ y $v \in C_j$. Entonces $f(C_i) > f(C_j)$.

Demostración: Lo podemos encontrar en el libro "Introducción a los Algoritmos." de T.H. Cormen.

El corolario de este lema es obvio.

Sean C_i y C_j dos componentes fuertemente conexos distintos en el grafo $G = (V, A)$. Suponga que existe una arista $(u, v) \in A^t$, donde $u \in C_i$ y $v \in C_j$. Entonces $f(C_i) < f(C_j)$.

Demostración: Lo podemos encontrar en el libro "Introducción a los Algoritmos." de T.H. Cormen.

El corolario nos proporciona la clave para comprender por qué funciona el algoritmo de Kosaraju. Examinemos lo que sucede cuando realizamos la segunda búsqueda en profundidad, que actúa sobre G^T . Comenzamos con el componente C fuertemente conexo cuyo tiempo de finalización $f(C)$ es máximo. La búsqueda comienza desde algún vértice $x \in C$, y visita todos los vértices en C . Según el Corolario, G^T no contiene aristas de C a ningún otro componente fuertemente conexo, por lo que la búsqueda desde x no visitará vértices en ningún otro componente. Por lo tanto, el árbol enraizado en x contiene exactamente los vértices de C . Después de completar la visita a todos los vértices en C , la búsqueda selecciona como raíz un vértice de algún otro componente C_i fuertemente conexo cuyo tiempo de finalización $f(C_i)$ es máximo en todos los componentes que no sean C . Nuevamente, la búsqueda visitará todos los vértices en C_i , pero según el Corolario, las únicas aristas en G^T desde C_i a cualquier otro componente deben ser a C , que ya hemos visitado. En general, cuando la búsqueda de profundidad de G^T visita cualquier componente fuertemente conexo, cualquier arista fuera de ese componente debe ser a los componentes que la búsqueda ya visitó. Por lo tanto, cada árbol de profundidad, será exactamente un componente fuertemente conexo. El siguiente teorema formaliza este argumento.

El algoritmo de Kosaraju calcula correctamente los componentes fuertemente conexos de un grafo dirigido G .

Demostración: Lo podemos encontrar en el libro "Introducción a los Algoritmos." de T.H. Cormen.

Algoritmos Greedy.



Elementos de una estrategia greedy

Un algoritmo codicioso obtiene una solución óptima a un problema haciendo una secuencia de elecciones. En cada punto de decisión, el algoritmo toma la decisión que parece mejor en este momento. Esta estrategia heurística no siempre produce una solución óptima, pero a veces lo hace. Esta sección discute algunas de las propiedades generales de los métodos codiciosos.

Propiedad de elección codiciosa.

El primer ingrediente clave es la propiedad de elección codiciosa: **podemos armar una solución globalmente óptima haciendo elecciones localmente óptimas (codiciosas)**. En otras palabras, cuando consideramos qué opción tomar, hacemos la elección que mejor se ve en el problema actual, sin considerar los resultados de los subproblemas (tampoco consideramos como llegamos a ese problema en particular, ni como se resolverá a posterior).

Aquí es donde los algoritmos codiciosos difieren de la programación dinámica. En la programación dinámica, hacemos una elección en cada paso, pero la elección generalmente depende de las soluciones a los subproblemas. En consecuencia, normalmente resolvemos problemas de programación dinámica de manera ascendente, pasando de subproblemas más pequeños a subproblemas más grandes.

(Alternativamente, podemos resolverlos de arriba hacia abajo, pero memorizando. Por supuesto, aunque el código funciona de arriba hacia abajo, aún debemos resolver los subproblemas antes de tomar una decisión). En un algoritmo codicioso, hacemos cualquier elección que parezca mejor en el momento y luego resuelve el subproblema que queda. La elección realizada por un algoritmo codicioso puede depender de elecciones realizadas con anterioridad, pero no puede depender de ninguna elección futura o de las soluciones a subproblemas. **Por lo tanto, a diferencia de la programación dinámica, que resuelve los subproblemas antes de hacer la primera elección, un algoritmo codicioso hace su primera elección antes de resolver cualquier subproblema.** Un algoritmo de programación dinámica avanza de abajo hacia arriba, mientras que una estrategia ambiciosa generalmente progresar de arriba hacia abajo, haciendo una elección codiciosa tras otra, reduciendo cada instancia problemática dada a una más pequeña.

Por supuesto, **debemos demostrar que una elección codiciosa en cada paso produce una solución globalmente óptima.** Luego muestra cómo modificar la solución para sustituir la opción codiciosa por otra opción, lo que resulta en un subproblema similar, pero más pequeño.

Por lo general, podemos hacer la elección codiciosa de manera más eficiente que cuando tenemos que considerar un conjunto más amplio de opciones.

Subestructura optima.

Un problema exhibe una subestructura óptima si una solución óptima al problema contiene soluciones óptimas a subproblemas. Esta propiedad es un ingrediente clave para evaluar la aplicabilidad tanto de la programación dinámica, como de los algoritmos codiciosos.

Usualmente usamos un enfoque más directo con respecto a la subestructura óptima cuando la aplicamos a algoritmos codiciosos. Tenemos el lujo de asumir que llegamos a un subproblema al haber tomado la codiciosa elección en el problema original. **Todo lo que realmente necesitamos hacer es argumentar que una solución óptima al subproblema, combinada con la codiciosa elección ya realizada, produce una solución óptima al problema original.** Este esquema utiliza implícitamente la inducción en los subproblemas para demostrar que tomar la decisión codiciosa en cada paso produce una solución óptima.

El proceso que seguimos para desarrollar un algoritmo codicioso realiza los siguientes pasos:

1. Determinar la subestructura optima de el problema.
2. Desarrollar una solución recursiva.
3. Muestre que si hacemos la elección codiciosa, solo queda un subproblema.
4. Demuestre que siempre es seguro tomar la decisión codiciosa. (Los pasos 3 y 4 pueden ocurrir en cualquier orden).
5. Desarrolle un algoritmo recursivo que implemente la estrategia codiciosa.
6. Convierta el algoritmo recursivo en un algoritmo iterativo.

Sin embargo, estos son los que generalmente funcionan bien:

1. Transmita el problema de optimización como uno **en el que hacemos una elección y nos queda un subproblema para resolver.**
2. Demuestre que siempre **hay una solución óptima (teórica) para el problema original obtenida construida en base a decisiones codiciosas, de modo que la elección codiciosa sea siempre segura.**
3. Demuestre una subestructura óptima demostrando que, **después de haber tomado la decisión codiciosa, lo que queda es un subproblema con la propiedad de que si combinamos una solución óptima al subproblema con**

la elección codiciosa que hemos hecho, llegamos a una solución óptima al original problema. A esto lo podemos llamar inducción en cierto punto.

Nota: Los items antes propuestos anteriormente no son pasos, son consejos que podemos tomar para alcanzar una solución greedy.

Utilizaremos este proceso más directo en secciones posteriores de este capítulo. Sin embargo, debajo de cada algoritmo codicioso, casi siempre hay una solución de programación dinámica más engorrosa.

¿Cómo podemos saber si un algoritmo codicioso resolverá un problema de optimización particular? De ninguna manera funciona todo el tiempo, pero la propiedad de elección codiciosa y la subestructura óptima son los dos ingredientes clave. Si podemos demostrar que el problema tiene estas propiedades, entonces estamos en camino de desarrollar un algoritmo codicioso para ello.

Interval Scheduling.

Enunciado.

Tenemos un conjunto de solicitudes $1, 2, \dots, n$; la i -ésima solicitud corresponde a un intervalo de tiempo que comienza en $s(i)$ y termina en $f(i)$. Diré que un subconjunto de las solicitudes es compatible si no se superponen dos de ellas a tiempo, y nuestro objetivo es aceptar un subconjunto compatible tan grande como sea posible. Los conjuntos compatibles de tamaño máximo se denominarán óptimos.

Nota: en este enunciado ya se da primer paso de nuestra guía para la resolución de problemas greedy. Se nos presenta un caso de optimización en donde se debe hallar el conjunto de solicitudes compatibles de tamaño máximo.

Diseño de la solución.

Una regla codiciosa que conduce a la solución óptima se basa en la siguiente idea: primero debemos aceptar la solicitud que finaliza primero, es decir, la solicitud i para la cual $f(i)$ es lo más pequeña posible. Esta también es una idea bastante natural: nos aseguramos de que nuestro recurso se vuelva libre lo antes posible y al mismo tiempo satisface una solicitud. De esta manera podemos maximizar el tiempo restante para satisfacer otras solicitudes.

Nota: Con esto, completamos el primer paso para la resolución greedy. Expresamos el problema como un “problema en el que hacemos una elección codiciosa y nos queda un subproblema por resolver”.

Partimos de un problema original y tras cada elección, añadiremos una y solo una solicitud a nuestro conjunto de solución, y nos quedara un subconjunto de solicitudes sin analizar.

Expongamos el algoritmo un poco más formalmente. Usaremos R para denotar el conjunto de solicitudes que aún no hemos aceptado ni rechazado, y utilizaremos A para denotar el conjunto de solicitudes aceptadas.

Inicialmente, deje que R sea el conjunto de todas las solicitudes, y deje que A esté vacío.

Mientras R aún no está vacío:

 Elija una solicitud i en R que tenga el menor tiempo de acabado.

 Añadir solicitud i a A .

 Eliminar todas las solicitudes de R que no sean compatibles con la solicitud i .

Devuelve el conjunto A como el conjunto de solicitudes aceptadas.

Análisis de la solución.

Nota: Aquí no solo intentamos demostrar que la solución es óptima, sino que nuestro algoritmo, basado en las dos propiedades de las estrategias greedy, es capaz de obtener un solución óptima.

Lo que necesitamos mostrar es que esta solución es óptima. Entonces, para fines de comparación, dejemos que O sea un conjunto óptimo de intervalos. Idealmente, uno podría querer mostrar que $A = O$, pero esto es demasiado pedir: puede haber muchas soluciones óptimas, y en el mejor de los casos A es igual a una sola de ellas. Entonces, simplemente mostraremos que $|A| = |O|$, es decir, que A contiene el mismo número de intervalos que O y, por lo tanto, también es una solución óptima.

La idea subyacente a la prueba, como sugerimos inicialmente, será encontrar un sentido en el que nuestro algoritmo codicioso "se mantenga por delante" (stays ahead) de esta solución O . **Compararemos las soluciones parciales que el algoritmo codicioso construye con segmentos iniciales de la solución O , y demuestre que el algoritmo codicioso está mejorando paso a paso.**

Introducimos alguna notación para ayudar con esta prueba. Sea i_1, \dots, i_k ser el conjunto de solicitudes en A en el orden en que se agregaron a A . Tenga en cuenta que $|A| = k$. Del mismo modo, sea el conjunto de solicitudes en O se denote por j_1, \dots, j_m . Nuestro objetivo es demostrar que $k = m$. Suponga que las solicitudes en O también se ordenan en el orden natural de izquierda a derecha de los intervalos correspondientes, es decir, en el orden de los puntos de inicio y finalización. Tenga en cuenta que las solicitudes en O son compatibles, lo que implica que los puntos de inicio tienen el mismo orden que los puntos de finalización.

Nuestra intuición para el método codicioso vino de querer que nuestro recurso volviera a ser libre lo antes posible después de satisfacer la primera solicitud. Y, de hecho, **nuestra regla codiciosa garantiza que $f(i_1) \leq f(j_1)$** . Este es el sentido en el que queremos mostrar que nuestra codiciosa regla "se mantiene por delante", que cada uno de sus intervalos termina al menos tan pronto como el intervalo correspondiente en el conjunto O . Por lo tanto, ahora demostramos que para cada $r \geq 1$, la r^{th} solicitud aceptada en la programación del algoritmo finaliza a más tardar en la solicitud solicitada en la programación óptima.

Lema: Para todos los indices $r \leq k$, tenemos que $f(i_r) \leq f(j_r)$.

Demostración: Probamos esta afirmación por inducción. Para $r = 1$, la afirmación es claramente cierta: el algoritmo comienza seleccionando la solicitud i_1 con un tiempo de finalización mínimo.

Ahora supongamos que $r > 1$. Asumiremos como nuestra hipótesis de inducción que el enunciado es verdadero para $r - 1$, y trataremos de demostrarlo para r . La hipótesis de inducción nos permite suponer que $f(i_{r-1}) \leq f(j_{r-1})$. Para que el intervalo r^{th} del algoritmo no termine antes también, necesitaría "quedarse atrás" como se muestra. Pero hay una razón simple por la que esto no podría suceder: en lugar de elegir un intervalo de finalización posterior, el algoritmo codicioso siempre tiene la opción (en el peor de los casos) de elegir j_r y así cumplir con el paso de inducción.

Se sabe que en el conjunto óptimo O todos los intervalos son compatibles, lo que significa que $f(j_{r-1}) \leq s(j_r)$. Combinando esto con la hipótesis inductiva $f(i_{r-1}) \leq f(j_{r-1})$, llegamos a que $f(i_{r-1}) \leq s(j_r)$. Así, el intervalo j_r está en el conjunto R de intervalos disponibles en el momento en que el codicioso algoritmo selecciona i_r . El algoritmo codicioso selecciona el intervalo disponible con el tiempo de finalización más pequeño; Como el intervalo j_r es uno de estos intervalos disponibles, tenemos $f(i_r) \leq f(j_r)$. Con esto se completa la etapa de inducción.

Nota: Este lema denota la subestructura óptima del problema y que nuestra elección greedy es siempre segura, ya que en cada paso se acerca a la solución óptima teórica.

Nota: Ponerse “por delante” de la solución optima, es un recurso muy importante dentro de la programación greedy. En palabras cortas, quiere decir, que bajo nuestros términos o reglas o decisiones, la solución que propongamos siempre es mejor o iguala a la solución optima.

Ahora mostraremos que el algoritmo retorna siempre un conjunto optimo.

Lema: El algoritmo greedy siempre retorna un conjunto optimo A.

Demostración: Probamos la declaración por contradicción. Si A no es óptimo, entonces un conjunto óptimo O debe tener más solicitudes, es decir, debemos tener $m > k$. Aplicando lemas anteriores, con $r = k$, obtenemos que $f(i_k) \leq f(j_k)$. Como $m > k$, hay una solicitud j_{k+1} en O. Esta solicitud comienza después de que finaliza la solicitud j_k y, por lo tanto, después de que i_k finaliza. Entonces, después de eliminar todas las solicitudes que no son compatibles con las solicitudes i_1, \dots, i_k , el conjunto de posibles solicitudes R todavía contiene j_{k+1} . Pero el algoritmo codicioso se detiene con la solicitud i_k , y solo se supone que se detiene cuando R está vacío, una contradicción.

Podemos hacer que nuestro algoritmo se ejecute en el tiempo $O(n\log n)$ de la siguiente manera. Comenzamos ordenando las n solicitudes en orden de tiempo de finalización y etiquetándolas en este orden; es decir, asumiremos que $f(i) \leq f(j)$ cuando $i \leq j$. Esto lleva tiempo $O(n\log n)$.

En un tiempo adicional de $O(n)$, construimos una matriz $S[1 \dots n]$ con la propiedad de que $S[i]$ contiene el valor $s(i)$. Ahora seleccionamos solicitudes procesando los intervalos en orden creciente de $f(i)$. Siempre seleccionamos el primer intervalo; luego iteramos a través de los intervalos en orden hasta alcanzar el primer intervalo j para el cual $s(j) \geq f(1)$; luego seleccionamos este también. De manera más general, si el intervalo más reciente que hemos seleccionado finaliza en el tiempo f , continuamos iterando a través de intervalos subsiguientes hasta que llegamos al primer j para el cual $s(j) \geq f$. De esta manera, implementamos el algoritmo codicioso analizado anteriormente en una pasada a través de los intervalos, pasando un tiempo constante por intervalo. Por lo tanto, esta parte del algoritmo lleva tiempo $O(n)$

Queremos apuntar ciertas cuestiones:

- En cada uno de nuestros lemas, hacemos alusión a una solución optima teórica. Y lo que pretendemos mostrar es que mediante nuestro algoritmo, se puede llegar a una solución equivalente. Esta estrategia es mas que bienvenida en los algoritmos greedy.
- Dada una solución optima teórica, ¿como se compara con una solución obtenida a través de nuestro algoritmo? esa es la pregunta clave que nos tenemos que hacer, para luego, plantear las propiedades greedy del problema.
- Algo que no se menciona es si el algoritmo genera una solución compatible, se da por hecho. Pues bueno, es importante demostrarlo, inclusive antes de comenzar con el resto de las demostraciones.

Video Stream

Enunciado

Tenemos un conjunto $\{1, 2, 3, \dots, n\}$ videos que se deben enviar por un canal de comunicación. Cada video cuenta con un tamaño b_i de bits y una duración t_i de segundos (tasa constante).

El canal de comunicación permite enviar un solo video a la vez y no permite interrupciones entre videos. Por otro lado, la cantidad de bits enviados desde el momento 0 hasta el momento t no puede superar un valor acumulado de $r \times t$ bits, siendo r un valor en bits.

¿Existe un orden valido para enviar los videos? De ser así, generar el orden.

Diseño de algoritmo

Como bien plantea la guia para la resolución de problemas greedy, lo primero que debemos intentar es plantear el problema como un problema de optimización en el que hacemos una elección y nos queda un subproblema por resolver.

En este sentido podemos proponer:

```
Sea un conjunto V={1, 2, 3, ..., n} de videos.  
Sea un conjunto ordenado vacio O de videos.  
Mientras V sea distinto de vacio:  
    Seleccionar un v de V de acuerdo a una elección greedy.  
    Agregar v a O.  
Si en algun instante t, la cantidad de bits enviados supera a r veces t:  
    No existe una programación válida.  
Sino:  
    Existe una programación válida y es O.
```

Y en cuanto al criterio greedy se nos puede ocurrir muchísimas ideas e ir probando cada una de ellas.

Una posible idea es plantear el problema en términos de optimización. Si el canal de comunicación es nuestro recurso, lo que vamos a querer es liberar el mismo lo antes posible. Entonces seleccionaremos al video que consuma la menor cantidad de bits por segundo o dicho de otra forma, cuyo bit rate $br_i = b_i/t_i$ sea el menor de todos.

Y esto funciona, pero vamos un poco mas lejos. Llamaremos resto R_i a la diferencia entre $r \times t_i$ y b_i .

- Si el $R_i \geq 0$ lo llamaremos **credito**.
- Si el $R_i < 0$ lo llamaremos **debito**.

Una clara consecuencia de este criterio es que acumularemos crédito al inicio, y luego iremos consumiendo ese crédito.

Formalicemos todo esto en un algoritmo.

```
Ordenar los videos según bit rate de menor a mayor en un arreglo V.  
Credito = 0.  
Sea P un arreglo ordenado vacío.  
Mientras credito sea mayor o igual a 0 y V no sea vacío:  
    Sea un v de V.  
    Credito += bitrate(v).  
    Agregar v a P.  
  
    Si credito es mayor o igual a 0:  
        Entonces hay una programación válida y es P.  
    Sino:  
        No hay una programación válida.
```

Analisis de la solución

Debido al ordenamiento, ya tenemos que contar con un orden de complejidad de $O(n\log n)$. El resto del algoritmo es lineal, dando por resultado una complejidad temporal de $O(n\log n)$. Pero, podemos hacerlo mejor.

La realidad es que para saber si existe o no una programación válida, solo debemos acumular todos los restos y si el resultado es positivo, entonces existe una programación válida, sino no.

Entonces, saber si existe o no una programación válida, nos tomara $O(n)$.

Y la pregunta que nos hacemos es ¿Necesitamos el ordenamiento? ¿Es condición necesaria para obtener una solución? Si bien, al principio planteamos el problema como un problema de optimización, este no es el caso, acá se trata de obtener una de las tantas soluciones posibles.

La realidad es que no es necesario el ordenamiento. Basta con agregar primero los vídeos cuyo resto sea positivo (que nos de crédito) y luego, los de resto negativo (que consuman el crédito que obtuvimos). Y esto lo podemos en $O(n)$.

The Minimum Spanning Tree

Enunciado.

Supongamos que tenemos un conjunto de ubicaciones $V = v_1, v_2, \dots, v_n$, y queremos construir una red de comunicación sobre ellas. La red debe estar conectada, debe haber una ruta entre cada par de nodos, pero sujeto a este requisito, deseamos construirla de la manera más económica posible.

Para ciertos pares (v_i, v_j) , podemos construir un enlace directo entre v_i y v_j por un cierto costo $c(v_i, v_j) > 0$. Por lo tanto, podemos representar el conjunto de enlaces posibles que se pueden construir usando un grafo $G = (V, E)$, con un costo positivo c_e asociado con cada arista $e = (v_i, v_j)$. El problema es encontrar un subconjunto de los aristas $T \subseteq E$ para que el grafo (V, T) esté conectado y el costo total $\sum_{e \in T} c_e$ sea lo más pequeño posible. (Asumiremos que el grafo completo G está conectado; de lo contrario, no hay solución posible).

Aquí hay una observación básica.

Lema: Sea T una solución de costo mínimo para el problema de diseño de red definido anteriormente. Entonces (V, T) es un árbol.

Demostración: Por definición, (V, T) debe estar conectado; mostramos que tampoco contendrá ciclos. De hecho, suponga que contiene un ciclo C , y deje que e sea cualquier arista en C . Afirmamos que $(V, T - e)$ todavía está conectado, ya que cualquier camino que anteriormente usaba el arista e ahora puede ir "a lo largo" del resto del ciclo C en su lugar. De ello se deduce que $(V, T - e)$ también es una solución válida para el problema, y es más barato, una contradicción.

Si permitimos que algunas aristas tengan un costo 0 (es decir, asumimos solo que los costos c_e no son negativos), entonces una solución de costo mínimo para el problema del diseño de la red puede tener aristas adicionales, aristas que tienen un costo 0 y podrían eliminarse opcionalmente. Pero incluso en este caso, siempre hay una solución de costo mínimo que es un árbol. A partir de cualquier solución óptima, podríamos seguir eliminando aristas en los ciclos hasta que tengamos un árbol; con aristas no negativos, el costo no aumentaría durante este proceso.

Llamaremos a un subconjunto $T \subseteq E$ un árbol de expansión de G si (V, T) es un árbol. La declaración anterior dice que el objetivo de nuestro problema de diseño de red puede reformularse como encontrar el árbol de expansión más barato del grafo; por esta razón, generalmente se llama el problema del árbol de expansión mínimo. A menos que G sea un grafo muy simple, tendrá exponencialmente muchos árboles de expansión diferentes, cuyas estructuras pueden verse muy diferentes entre sí. Por lo tanto, no está nada claro cómo encontrar eficientemente el árbol más barato entre todas estas opciones.

Diseñando el algoritmo.

Aquí hay tres algoritmos codiciosos, cada uno de los cuales encuentra correctamente un árbol de expansión mínimo.

- Un algoritmo simple comienza sin ningún arista y construye un árbol de expansión insertando aristas sucesivamente desde E en orden de costo creciente. A medida que avanzamos por las aristas en este orden, insertamos cada arista e siempre que no cree un ciclo cuando se agrega a las aristas que ya hemos insertado. Si, por otro lado, insertando e resultaría en un ciclo, entonces simplemente descartamos e y continuamos. Este enfoque se llama Algoritmo de Kruskal.

- Se puede diseñar otro algoritmo codicioso simple por analogía con el Algoritmo de Dijkstra para rutas, aunque, de hecho, es aún más simple de especificar que el Algoritmo de Dijkstra. Comenzamos con un nodo raíz s e intentamos crecer con avidez un árbol desde afuera. En cada paso, simplemente agregamos el nodo que se puede conectar tan barato como sea posible al árbol parcial que ya tenemos. Más concretamente, mantenemos un conjunto $S \subseteq V$ en el que se ha construido un árbol de expansión hasta ahora. Inicialmente, $S = s$. En cada iteración, crecemos S en un nodo, agregando el nodo v que minimiza el "costo de fijación" $\min_{e=(u,v):u \in S} c_e$, e incluyendo la arista $e = (u, v)$ que logra este mínimo en el árbol de expansión. Este enfoque se llama Algoritmo de Prim.
- Finalmente, podemos diseñar un algoritmo codicioso ejecutando una especie de versión "hacia atrás" del algoritmo de Kruskal. Específicamente, comenzamos con el gráfico completo (V, E) y comenzamos a eliminar aristas en orden decreciente de costo. A medida que llegamos a cada arista e (comenzando por el más costoso), lo eliminamos siempre que hacerlo no desconecte realmente el gráfico que tenemos actualmente. Por falta de un nombre mejor, este enfoque generalmente se llama Algoritmo de eliminación inversa (por lo que podemos decir, nunca ha sido nombrado por una persona específica).

Análisis.

¿Cuándo es seguro incluir una arista en el árbol de expansión mínimo? El hecho crucial sobre la inserción del arista es la siguiente declaración, a la que nos referiremos como la **propiedad de corte**.

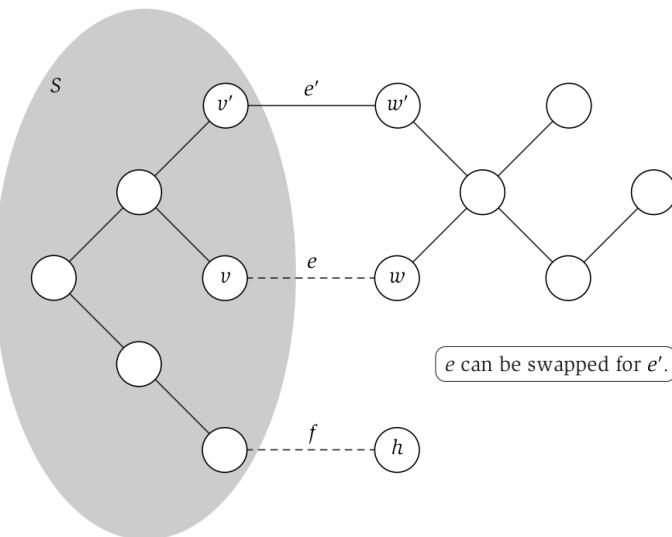


Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T , as described in the proof of (4.17).

Lema: Suponga que todos los costos de aristas son distintos. Supongamos que S sea cualquier subconjunto de nodos que no esté vacío ni sea igual a todos los de V , y que la arista $e = (v, w)$ sea la arista de costo mínimo con un extremo en S y el otro en $V - S$. Entonces, cada árbol mínimo de expansión contiene la arista e .

Demostración: Sea T un árbol de expansión que no contenga e ; Necesitamos demostrar que T no tiene el mínimo costo posible. Haremos esto usando un argumento de intercambio: identificaremos una arista e^* en T que es más costosa que e , y con la propiedad que intercambia e por e^* resulta en otro árbol de expansión. Este árbol de expansión resultante será más barato que T , como se deseé. El meollo de la cuestión es, por lo tanto, encontrar una arista que se pueda intercambiar con éxito con e . Recuerde que los extremos de e son v y w . T es un árbol de expansión, por lo que debe haber una ruta P en T de v a w . Comenzando en v , supongamos que seguimos los nodos de P en secuencia; hay un primer nodo w^* en P que está en $V - S$. Sea $v^* \in S$ el nodo justo antes de w^* en P , y sea $e^* = (v^*, w^*)$ la arista que los une. Por lo tanto, e^* es una arista de T con un extremo en S y el otro en $V - S$.

Si cambiamos e por e^* , obtenemos un conjunto de aristas $T' = T - \{e\} \cup \{e^*\}$. Afirmando que T' es un árbol de expansión. Claramente (V, T') está conectado, ya

que (V, T) está conectado, y cualquier ruta en (V, T) que usara el arista $e^* = (v^*, w^*)$ ahora se puede "redirigir" en (V, T^*) para seguir la porción de P de v^* a v , luego la arista e , y luego la porción de P de w a w^* . Para ver que (V, T^*) también es acíclico, tenga en cuenta que el único ciclo en $(V, T^* \cup e^*)$ es el compuesto por e y la ruta P , y este ciclo no está presente en (V, T^*) debido a la eliminación de e^* .

Notamos anteriormente que el arista e^* tiene un extremo en S y el otro en $V - S$. Pero e es la arista más barata con esta propiedad, y entonces $c_e < c_{e^*}$. (La desigualdad es estricta ya que no hay dos aristas que tengan el mismo costo). Por lo tanto, el costo total de T^* es menor que el de T , como se desea.

Existen tres lemas que claramente apuntan a decir que los algoritmos antes mencionados funcionan correctamente.

Lema: El algoritmo de Kruskal produce un árbol de expansión mínima de G .

Demostración: Consideré cualquier arista $e = (v, w)$ agregado por el algoritmo de Kruskal, y deje que S sea el conjunto de todos los nodos a los cuales v tiene una ruta en el momento justo antes de que se agregue e . Claramente $v \in S$, pero $w \notin S$, ya que agregar e no crea un ciclo. Además, todavía no se ha encontrado ningún arista de S a $V - S$, ya que dicha arista podría haberse agregado sin crear un ciclo, y por lo tanto habría sido agregado por el Algoritmo de Kruskal. Por lo tanto, e es el arista más barata con un extremo en S y el otro en $V - S$, y por lo tanto pertenece a cada árbol de expansión mínimo.

Entonces, si podemos demostrar que la salida (V, T) del algoritmo de Kruskal es, de hecho, un árbol de expansión de G , entonces habremos terminado. Claramente (V, T) no contiene ciclos, ya que el algoritmo está diseñado explícitamente para evitar la creación de ciclos. Además, si (V, T) no estuviera conectado, entonces existiría un subconjunto no vacío de nodos S (no igual a todos los V) de modo que no haya un arista de S a $V - S$. Pero esto contradice el comportamiento del algoritmo: sabemos que dado que G está conectado, hay al menos un arista entre S y $V - S$, y el algoritmo agregará el primero de estos que encuentre.

Lema: El algoritmo de Prim produce un árbol de expansión mínima de G .

Demostración: Para el algoritmo de Prim, también es muy fácil demostrar que solo agrega aristas que pertenecen a cada árbol de expansión mínimo. De hecho, en cada iteración del algoritmo, hay un conjunto $S \subseteq V$ en el que se ha construido un árbol de expansión parcial, y se agregan un nodo v y un arista e que minimizan la cantidad $mine = (u, v): u \in S \text{ y } e$. Por definición, e es el arista más barato con un extremo en S y el otro extremo en $V - S$, por lo que, según la propiedad de corte (4.17), se encuentra en cada árbol de expansión mínimo.

También es sencillo mostrar que el Algoritmo de Prim produce un árbol de expansión de G y, por lo tanto, produce un árbol de expansión mínimo.

¿Cuándo podemos garantizar que un arista no esté en el árbol de expansión mínimo? El hecho crucial sobre la eliminación de aristas es la siguiente declaración, a la que nos referiremos como la **propiedad del ciclo**.

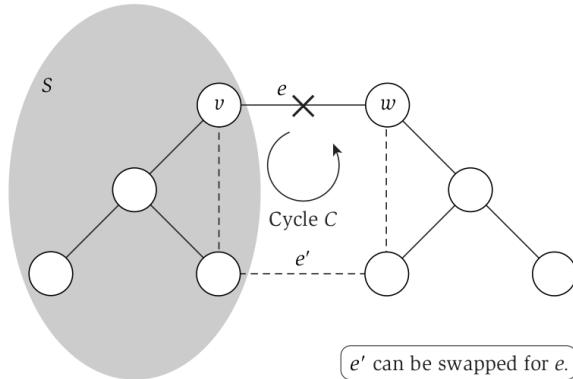


Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T , as described in the proof of (4.20).

Lema: Suponga que todos los costos de arista son distintos. Sea C cualquier ciclo en G , y sea el arista $e = (v, w)$ el arista más costoso que pertenece a C . Entonces e no pertenece a ningún árbol de expansión mínimo de G .

Demostración: Sea T un árbol de expansión que contiene e ; Necesitamos demostrar que T no tiene el mínimo costo posible. Por analogía con la prueba de la propiedad de corte, haremos esto con un argumento de intercambio, intercambiando e por un arista más barato de tal manera que todavía tengamos un árbol de expansión.

Entonces, la pregunta es: ¿cómo encontramos una arista más barata que se pueda intercambiar de esta manera con e ? Comencemos eliminando e de T ; esto divide los nodos en dos componentes: S , que contiene el nodo v ; y $V - S$, que contiene el nodo w . Ahora, la arista que usamos en lugar de e debe tener un extremo en S y el otro en $V - S$, para unir nuevamente el árbol.

Podemos encontrar tal arista siguiendo el ciclo C . Los aristas de C distintos de e forman, por definición, una ruta P con un extremo en v y el otro en w . Si seguimos P de v a w , comenzamos en S y terminamos en $V - S$, por lo que hay algún arista e' en P que cruza de S a $V - S$. Consulte la Figura 4.11 para ver una ilustración de esto. Ahora considere el conjunto de aristas $T' = T - \{e\} \cup \{e'\}$. Argumentando al igual que en la prueba de la propiedad de corte (4.17), el gráfico (V, T') está conectado y no tiene ciclos, por lo que T' es un árbol de expansión de G . Además, dado que e' es el arista más barato en el ciclo C , y e' pertenece a C , debe ser que e' es más barato que e , y por lo tanto T' es más barato que T , según se deseé.

Ahora que tenemos la propiedad de ciclo, es fácil demostrar que el algoritmo de eliminación inversa produce un árbol de expansión mínimo. La idea básica es análoga a las pruebas de optimización para los dos algoritmos anteriores: Reverse-Delete solo agrega una ventaja cuando está justificado por la propiedad de ciclo.

Lema: El algoritmo de eliminación inversa produce un árbol de expansión mínimo de G .

Demostración: Considere cualquier arista $e = (v, w)$ eliminado por Reverse-Delete. En el momento en que se elimina e , se encuentra en un ciclo C ; y dado que es el primer arista encontrado por el algoritmo en orden decreciente de costos de arista, debe ser el arista más costoso en C . Por lo tanto, en (4.20), e no pertenece a ningún árbol de expansión mínimo.

Entonces, si mostramos que la salida (V, T) de Reverse-Delete es un árbol de expansión de G , habremos terminado. Claramente (V, T) está conectado, ya que el algoritmo nunca elimina un arista cuando esto desconectará el gráfico. Ahora, supongamos por contradicción que (V, T) contiene un ciclo C . Considere la ventaja más costosa e en C , que sería la primera encontrada por el algoritmo. Este arista debería haberse eliminado, ya que su eliminación no habría desconectado el gráfico, y esto contradice el comportamiento de Reverse-Delete.

Códigos Huffman y compresión de datos

Enunciado

La compresión de datos es un campo mas que importante dentro de las ciencias de la computación, y en este caso, intentaremos obtener el mejor algoritmo de compresión con códigos prefijos.

Los códigos son convenciones de como representar cada mensaje utilizando una combinación de símbolos. Puede haber códigos de longitud variable o fija.

Llamaremos códigos decodificables a cualquiera que, dada un sucesión de códigos, solo existe un único mensaje valido. Los códigos de longitud fija son siempre decodificables, por dar un ejemplo, los códigos ASCII son códigos de longitud fija decodificable.

Por otra parte, están los códigos prefijos que también son siempre decodificables. Un código es prefijo si no existe ningún código que tenga un prefijo igual a otro código completo.

Entonces, lo que pretendemos es que dado un alfabeto $S = \{s_1, s_2, s_3 \dots s_n\}$, llamaremos $F = \{f_1, f_2, f_3, \dots, f_n\}$ a las frecuencia de cada letra del alfabeto S y $C(S) = \{c_1, c_2, c_3, \dots, c_n\}$ a los códigos prefijos asignados a cada letra de S .

Lo que intentamos lograr es que la longitud promedio de los códigos.

$$ABL(C(S)) = \sum_{i=1}^n f_i * |c_i|$$

Sea la menor para cualquier código prefijo $T(S)$. A esto lo llamaremos código prefijo optimo. Es importante entender que existen infinidad de códigos prefijos que cumplan con este requisito. Pero Huffman nos da un método para hallar un código prefijo optimo bajo estrategia greedy.

Diseño del algoritmo

El espacio de búsqueda de este problema es bastante complicado; incluye todas las formas posibles de mapear letras a cadenas de bits, sujeto a la propiedad definitoria de los códigos de prefijo. Para alfabetos que constan de un número extremadamente pequeño de letras, es factible buscar este espacio por la fuerza bruta, pero esto rápidamente se vuelve inviable.

Ahora describimos un método codicioso para construir un código de prefijo óptimo de manera muy eficiente. Como primer paso, es útil desarrollar un diagrama basado en árboles para representar códigos de prefijo que exponga su estructura más claramente.

Representación de códigos de prefijo usando árboles binarios. Supongamos que tomamos un árbol con raíz T en el que cada nodo que no es una hoja tiene como máximo dos hijos; a ese árbol lo llamamos árbol binario. Además, suponga que el número de hojas es igual al tamaño del alfabeto S , y etiquetamos cada hoja con una letra distinta en S .

Tal árbol binario etiquetado T naturalmente describe un código de prefijo, como sigue. Para cada letra $x \in S$, seguimos el camino desde la raíz hasta la hoja etiquetada x ; cada vez que la ruta va de un nodo a su hijo izquierdo, escribimos un 0, y cada vez que la ruta va de un nodo a su hijo derecho, escribimos un 1. Tomamos la cadena de bits resultante como la codificación de X .

Lema: El código construido en base a T es un código prefijo

La prueba de esto lo podrán encontrar en el libro “Diseño de algoritmos de Jon Kleinberg y Eva Tardos”

Esta relación entre el árbol binario y los códigos prefijos funciona perfectamente en el sentido contrario. Dado un código prefijo, podemos construir un árbol binario recursivamente. Empezamos por la raíz; todas las letras de S que empiecen con un 0, las dejamos en el sub árbol izquierdo, mientras que las que empiecen con un 1, las dejamos

en el sub árbol derecho. Luego, construimos los sub arboles, de manera recursiva, usando esta misma regla.

Lema: El árbol binario correspondiente a un código prefijo óptimo, es un árbol binario completo.

La prueba de esto lo podrán encontrar en el libro “Diseño de algoritmos de Jon Kleinberg y Eva Tardos”

Lema: Suponga que u y v son hojas de T^ , tales que $\text{profundidad}(u) < \text{profundidad}(v)$. Además, suponga que en un etiquetado de T^* corresponde a un **código de prefijo óptimo**, la hoja u es la etiqueta de $y \in S$ y la hoja v es la etiqueta de $z \in S$. Entonces $\text{frecuencia}(y) \geq \text{frecuencia}(z)$. Es decir, cuanto mas profunda es la hoja, mas largo es el código, pero menor la frecuencia de aparición*

La prueba de esto lo podrán encontrar en el libro “Diseño de algoritmos de Jon Kleinberg y Eva Tardos”

Con estos conceptos, es que presentamos el siguiente algoritmo.

```
Sea un alfabeto  $S$  con las respectivas frecuencias para cada letra.  
Sea Huffman la función que dado un alfabeto  $S$ :  
    Si  $S$  tiene dos letras, entonces:  
        Codificamos una letra con 0 y la otra con 1.  
        Terminamos.  
    Sean  $y^*$  e  $x^*$  las dos letras con menor frecuencia de  $S$ .  
    Construimos un nuevo alfabeto  $S^*$  eliminando  $y^*$  e  $x^*$  y  
    remplazandolas con una nueva letra  $z^*$  cuya frecuencia es la suma de  
    las frecuencias de  $x^*$  e  $y^*$ .  
    Ejecutamos recursivamente Huffman( $S^*$ ).
```

Análisis de la solución

Sea que ordenemos nuestro alfabeto de menor a mayor respecto a sus frecuencias, esto tomara un tiempo no menor a $O(n \log(n))$. Y el resto del algoritmo es lineal, por lo que el costo final en tiempo de ejecución es de $O(n \log(n))$.

Para probar que el algoritmo de Huffman es óptimo, tenemos que hacer dos demostraciones.

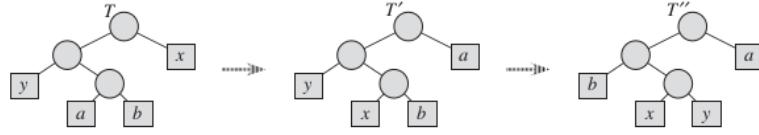
- Prueba de selección greedy: lograr demostrar que elegir en los dos códigos de menor peso nos acerca a la solución óptima.
- Prueba de sub problemas: Lograr demostrar que el sub problema derivado de nuestra elección se puede solucionar mediante la misma selección greedy.

Lema: Sea S un alfabeto en el que cada carácter $s \in S$ tiene frecuencia $s.freq$. Sean x e y dos caracteres en S que tienen las frecuencias más bajas. Entonces existe un código de prefijo óptimo para S en el que los códigos para x e y tienen la misma longitud y difieren solo en el último bit.

Demostración: La idea de la prueba es tomar el árbol T que representa un código de prefijo óptimo arbitrario y modificarlo para hacer un árbol que represente otro código de prefijo óptimo de modo que los caracteres x e y aparezcan como hojas hermanas de máxima profundidad en el nuevo árbol. Si podemos construir tal árbol, entonces las palabras de código para x e y tendrán la misma longitud y solo diferirán en el último bit.

Sea a y b ser dos caracteres que son hojas hermanas de máxima profundidad en T . Sin pérdida de generalidad, suponemos que $a.freq \leq b.freq$ y $x.freq \leq y.freq$. Dado que $x.freq$ e $y.freq$ son las dos frecuencias de hoja más bajas, en orden, y $a.freq$ y $b.freq$ son dos frecuencias arbitrarias, en orden, tenemos $x.freq \leq a.freq$ e $y.freq \leq b.freq$.

En el resto de la prueba, es posible que podamos tener $x. freq = a. freq$ o $y. freq = b. freq$. Sin embargo, si tuviéramos $x. freq = b. freq$, también tendríamos $a. freq = b. freq = x. freq = y. freq$, y el lema sería trivialmente verdadero. Por lo tanto, asumiremos que $x. freq \neq b. freq$, lo que significa que $x \neq b$.



Como se muestra en la figura, nosotros podemos cambiar de posición en T de a y x para producir un árbol T' , y cambiar b por y para producir el árbol T'' en el cual, x e y son hojas hermanas de máxima profundidad. Por la ecuación ABL, y los códigos $C(T)$ y $C(T')$ obtenidos a partir de los árboles correspondientes, podemos decir que la diferencia entre los árboles T y T es:

$$\begin{aligned}
ABL(C(T)) - ABL(C(T')) &= \sum_{i=1}^n f_i * |c(T)_i| - \sum_{i=1}^n f_i * |c(T')_i| \\
&= f_x * |c(T)_x| + f_a * |c(T)_a| - f_x * |c(T')_x| - f_a * |c(T')_a| \\
&= f_x * |c(T)_x| + f_a * |c(T)_a| - f_x * |c(T)_a| - f_a * |c(T)_x| \\
&= (f_a - f_x)(|c(T)_a| - |c(T)_x|) \\
&\geq 0
\end{aligned}$$

Por la ecuación mencionada anteriormente la diferencia entre los árboles T y T' es $ABL(C(T)) - ABL(C(T')) \geq 0$, porque tanto $a. freq - x. freq$ como $d_T(a) - d_T(x)$ no son negativos. Más específicamente, $a. freq - x. freq$ no es negativo porque x es una hoja de frecuencia mínima, y $|c(T)_a| - |c(T)_x|$ no es negativo porque a es una hoja de profundidad máxima en T .

Del mismo modo, el intercambio de y y b no aumenta el costo, por lo que $ABL(C(T')) - ABL(C(T'')) \geq 0$, porque tanto $a. freq - x. freq$ como $d_T(a) - d_T(x)$ no son negativos. Por lo tanto, $ABL(C(T'')) \leq ABL(C(T))$, y dado que T es óptimo, tenemos $ABL(C(T'')) \geq ABL(C(T))$, lo que implica $ABL(C(T'')) = ABL(C(T))$. Por lo tanto, T'' es un árbol óptimo en el que x e y aparecen como hojas hermanas de máxima profundidad, y por ende $C(T'')$ es un código prefijo óptimo, admitiendo así que la elección greedy de tomar los caracteres de menor frecuencia de S como base para construir un código prefijo óptimo, es siempre confiable.

Ahora vamos con el siguiente lema.

Lema: Sea S un alfabeto en el que cada carácter $s \in S$ tiene frecuencia $s. freq$. Sean x e y dos caracteres del alfabeto S con frecuencia mínima. Sea S' el alfabeto basado en S con los caracteres x e y removidos, y un nuevo carácter z agregado. Definimos $z. freq = x. freq + y. freq$. Sea T' el árbol representante de un código prefijo óptimo cualquiera para S' .

Entonces T obtenido a partir de T' reemplazando la hoja z por un nodo interno con x e y como hojas, representa un código prefijo óptimo para S .

Demostración: Se puede demostrar muy fácilmente que $ABL(C(T')) = ABL(C(T)) - x. freq - y. freq$, así que lo dejaremos como definición aunque no lo sea. Ahora probaremos el lema por contradicción.

Supongamos que T no es el árbol representativo de un código prefijo óptimo para S . Entonces existe un árbol T'' que si lo sea. Por ser tal, se tiene que cumplir que $ABL(C(T'')) \leq ABL(C(T))$. Sin perder generalidad y por el lema mencionado anteriormente, T'' tiene a x e y como hojas hermanas. Sea T''' un árbol igual a T'' con la excepción de que posee una hoja z que reemplaza a las hojas x e y de T'' y cuya frecuencia es igual a $z. freq = x. freq + y. freq$, entonces:

$$\begin{aligned}
ABL(C(T''')) &= ABL(C(T'')) - x.freq - y.freq \\
&< ABL(C(T)) - x.freq - y.freq \\
&= ABL(C(T'))
\end{aligned}$$

Entonces $ABL(C(T''')) \leq ABL(C(T'))$ lo cual entra en contradicción con nuestra hipótesis que dice que T' es árbol de un código prefijo óptimo. Entonces T si representa a un código prefijo óptimo.

Con esto queda demostrado que un sub problema derivado de nuestra elección se puede solucionar mediante la misma selección greedy.

Caminos mínimos en un Grafo.

Enunciado

En muchos casos, los grafos son usados como modelos de redes en las cuales, un viajero va de un punto a otro, atravesando una secuencia de lugares que están conectados entre sí. Como resultado, un problema algorítmico básico es determinar cuál es el camino más corto entre dos nodos. Incluso, se podría pedir más información y decir, dado un nodo de inicio s , cuál el camino más corto entre este y los demás nodos del grafo.

El enunciado concreto de el problema de los caminos cortos es el siguiente. Dado un grafo $G = (V, E)$, con un nodo s designado como nodo de inicio. Asumimos que s tiene un camino que lo une al resto de los nodos de G . Cada arista e tiene una longitud $l_e \geq 0$, indicando el tiempo (o la distancia, o el costo) que toma atravesar dicha arista. Para cada camino P , se puede calcular su largo, denotado como $l(P)$, como la suma de las longitudes de las aristas que componen este camino. Nuestro objetivo es determinar el camino más corto entre s y el resto de los nodos de G . Debemos mencionar que este problema es esta orientado a grafos dirigidos, aunque podemos aplicarlo sobre grafos no dirigidos, reemplazando cada una de las aristas no dirigidas, por dos aristas dirigidas con orientaciones opuestas y con la misma longitud.

Diseño del algoritmo

En 1959, Edsger Dijkstra propuso un algoritmo greedy muy simple para resolver el problema de los caminos cortos con un único nodo de origen. Comenzaremos por describir un algoritmo que solo determina el largo de los caminos cortos desde un nodo s al resto de los nodos. El algoritmo mantiene un conjunto S de vértices u para los cuales ya se determinó un camino corto cuya distancia desde s es $d(u)$. A este conjunto lo llamaremos conjuntos de nodos explorados. Inicialmente $S = s$, y $d(s) = 0$. Ahora, para cada nodo v de $V - S$, determinamos los caminos cortos que pueden ser construidos mediante los caminos cortos de los nodos ya explorados en S , seguidos por una arista (u, v) , donde u está en S , considerando que $d'(v) = \min_{e=(u,v); u \in S} d(u) + l_e$, a este lo llamaremos "factor de llegada". Y en base a este criterio es que elegimos que nodo de $v \in V - S$ se agregara a S .

```

Sea un grafo G, dirigido.
Sea S un conjunto de nodos explorados, inicialmente vacío.
Las distancias a cada nodo están inicializadas a un valor cercano al infinito.
Agregamos s a S, y asignamos distancia(s) = 0.

Mientras S sea distinto de V:
    Seleccionamos la arista e=(u, v) con u en S y v en V-S, cuyo factor de llegada igual a distancia(u) + longitud(e) sea la menor de todas.

    Agregamos v a S y definimos distancia(v) = distancia(u) +
    longitud(e)

```

Análisis de algoritmo

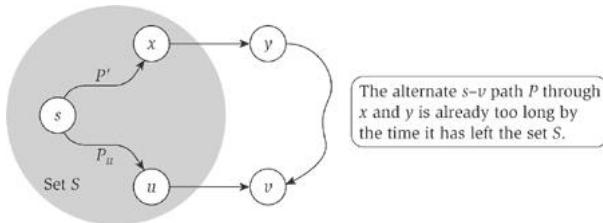
Primero analicemos si la elección greedy que hace el algoritmo es efectivo o no.

Lema: Considerando al considerando al conjunto S en cualquier punto del algoritmo en ejecución. Para cada $u \in S$, el camino P_u es el más corto entre s y u

Demostración: Demostramos esto por inducción sobre el tamaño de S . El caso $|S| = 1$ es fácil, ya que entonces tenemos $S = \{s\}$ y $d(s) = 0$. Supongamos que la afirmación se cumple cuando $|S| = k$ para algún valor de $k \geq 1$; ahora incrementamos S al tamaño $k + 1$ agregando el nodo v . Sea (u, v) la arista final en nuestra ruta P que une $s - v$

Por hipótesis de inducción, P_u es el camino su más corto para cada $u \in S$. Ahora considere cualquier otro camino P que une $s - v$; deseamos mostrar que es al menos tan largo como P_v . Para llegar a v , este camino P debe dejar el conjunto S en alguna parte; sea y el primer nodo en P que no está en S , y sea $x \in S$ el nodo justo antes de y .

La situación es ahora como se muestra en la figura, y la clave de la demostración es muy simple: P no puede ser más corto que P_v porque ya es al menos tan largo como P_v cuando ha dejado el conjunto S . De hecho, en la iteración $k + 1$, el algoritmo de Dijkstra debe haber considerado la adición de nodo y al conjunto S a través del arista (x, y) y rechazado esta opción a favor de añadir v . Esto significa que no hay camino de S a y a través de x que es más corto que P_v . Pero la sub ruta de P hasta y es dicha ruta, por lo que esta sub ruta es al menos tan larga como P_v . Dado que las longitudes de las aristas no son negativas, la trayectoria completa P es al menos tan larga como P_v también.



Esta es una prueba completa;

En cuanto a la complejidad algorítmica de esta solución, debemos tener en cuenta que el loop principal se ejecutara a lo sumo $n - 1$ veces, siendo n la cantidad de vértices o nodos.

Uno puede suponer que en cada iteración, se deben analizar las m aristas dando lugar a una complejidad de $O(nm)$, sin embargo, proponemos utilizar una cola de prioridad o heap para esta tarea, siendo el factor de llegada la clave para el orden de extracción de cada uno de los nodos. Este factor debe ser actualizado conforme se van descubriendo los distintos caminos. Cada operación de extracción e inserción tiene un costo $O(\log(n))$.

Actualizamos nuestro pseudocódigo para adecuarlo a esta nueva perspectiva.

```

Sea S el conjunto de nodos explorados.
Sea V el conjunto de nodos por explorar.
Sea n la cantidad de nodos en el grafo.
Sea Q una cola de prioridad de nodos ordenados por factor de llegada.

Agregamos s a S y definimos distancia(s) = 0.

Por cada nodo u vecino a s:
    Definimos factor(u) = longitud(e(s, u)).
    Definimos predecesor(u) = s.
    Agregamos u a Q.

Mientras S no tenga n nodos:
    Extraemos v de Q.
    Agregamos v a S.
    Actualizamos distancia(v) = factor(v)
    Por cada vecino w de v:
        Si w no esta en S:
            Definimos factor(w) = distancia(v) + longitud(v, w).
            Definimos predecesor(w) = v.
            Agragamos w a Q.

```

Ahora podemos decir que nuestra solución itera n , haciendo una extracción de Q con un costo $O(\log(n))$, además de hacer una inserción con el mismo costo por cada arista del nodo extraído, lo que en el peor de los casos se hace m veces. De esta forma, obtenemos un costo igual a $O((n + m)\log(n))$.

Mochila Fraccionaria.

Enunciado

Contamos con un contenedor que posee una capacidad máxima igual a W y una cantidad n de elementos fraccionables que tienen un cierto valor v_i y un peso (o volumen) igual a w_i . Como ejemplo, podemos decir que tenemos caramelos de distintos colores. En este caso, los caramelos de un mismo color es un elemento en si, con un determinado valor y volumen.

Queremos seleccionar un subconjunto de elementos o fracciones de ellos de modo que se maximice el valor almacenado y sin superar la capacidad de la mochila.

Diseño del algoritmo

Nuestra elección greedy se basara en priorizar los elementos más valiosos por unidad. Llenaremos el contenedor con la mayor cantidad posible de unidades del elemento disponible más valioso por unidad. Repetiremos el proceso mientras quede espacio en el contenedor y elementos disponibles. A continuación, mostramos el pseudocódigo de la solución.

```

Sea Lugar = w.
Sea Valor = 0.

Mientras existan elementos disponibles y Lugar > 0:
    Sea x el elemento disponible con mayor valor por unidad.
    Cantidad = min(wx, Lugar).
    Valor += Cantidad * vx.
    Lugar -= Cantidad.
    Quitar x de elementos disponibles.

Devolver Valor.

```

Análisis de la solución

Veamos si nuestra solución es óptima.

Lema: El algoritmo siempre devuelve una solución óptima.

Demostración: Sea $G = \{g_1, g_2, \dots, g_i\}$ el subconjunto obtenido por nuestra solución. Supongamos que existe $O = \{o_1, o_2, \dots, o_j\}$ un subconjunto óptimo tal que el valor acumulado de ambos conjuntos es tal que $\text{valor}(O) \geq \text{valor}(G)$. Para que esto ocurra, debe existir en O al menos un elemento o_k que no existe en G con mayor valor por unidad que al menos un elemento que si se encuentre en G , o esta en menor cantidad en G y esa diferencia está en al menos un elemento en O en menor valor.

Pero esto implicaría que no se siguieron los pasos correctamente para obtener G , lo cual es una contradicción. Por ende, no existe un subconjunto O tal que $\text{valor}(O) \geq \text{valor}(G)$

Respecto al orden de ejecución, ordenar los elementos tiene un costo de $O(n\log(n))$. Por otro lado, iteramos a lo sumo n veces. Esto da como resultado un algoritmo de $O(n\log(n))$.

The Blair Witch Project

Enunciado.

Suponga que tres de sus amigos, inspirados por las repetidas vistas del fenómeno de la película de terror The Blair Witch Project, han decidido caminar por el sendero de los Apalaches este verano. Quieren caminar tanto como sea posible por día pero, por razones obvias, no después del anochecer. En un mapa, han identificado un gran conjunto de buenos puntos de parada para acampar, y están considerando el siguiente sistema para decidir cuándo parar durante el día. Cada vez que llegan a un punto de parada potencial, determinan si pueden llegar al siguiente antes del anochecer. Si pueden lograrlo, entonces siguen caminando; de lo contrario, se detienen.

A pesar de muchos inconvenientes significativos, afirman que este sistema tiene una buena característica. "Dado que solo vamos de excursión a la luz del día", afirman, "minimiza la cantidad de paradas de campamento que tenemos que hacer".

¿Es esto cierto? El sistema propuesto es un algoritmo codicioso y deseamos determinar si minimiza el número de paradas necesarias.

Para hacer esta pregunta precisa, hagamos el siguiente conjunto de supuestos simplificadores. Modelaremos el Sendero de los Apalaches como un segmento de línea larga de longitud L , y asumiremos que sus amigos pueden caminar d millas por día (independientemente del terreno, las condiciones climáticas, etc.). Asumiremos que los puntos de parada potenciales se encuentran a distancias x_1, x_2, \dots, x_n desde el inicio del recorrido. También asumiremos (muy generosamente) que sus amigos siempre tienen la razón cuando estiman si pueden llegar al siguiente punto de parada antes del anochecer. Diremos que un conjunto de puntos de parada es válido si la distancia entre cada par adyacente es como máximo d , el primero está a distancia como máximo d desde el inicio del recorrido, y el último está a distancia como máximo d desde Fin del sendero. Por lo tanto, un conjunto de puntos de parada es válido si uno pudiera acampar solo en estos lugares y aún así cruzar todo el camino. Asumiremos, naturalmente, que el conjunto completo de n puntos de detención es válido; de lo contrario, no habría forma de hacerlo por completo.

Ahora podemos plantear la pregunta de la siguiente manera. ¿Es el algoritmo codicioso de tus amigos, ir de excursión el mayor tiempo posible cada día, óptimo, en el sentido de que encuentra un conjunto válido cuyo tamaño es lo más pequeño posible?

Análisis de la solución.

Para decir que el algoritmo brindado, es un algoritmo óptimo, debemos poder demostrar que la solución brindada en cada paso siempre mejora y que esta a la altura de una solución optima. O decirlo de otra forma, siempre esta a la cabeza.

Sea $R = \{x_{p1}, \dots, x_{pk}\}$ el conjunto de puntos de paradas obtenido por el algoritmo. Supongamos por contradicción, que existe un conjunto de puntos mas pequeño llamado $S = \{x_{q1}, \dots, x_{qm}\}$ con $m < k$.

Para obtener una contradicción, nosotros primero mostraremos que el punto de acampe alcanzado por el algoritmo greedy en un día j esta mas lejos que el punto de acampe alcanzado por una solución optima alternativa. Es decir:

Lema: Por cada $J = 1, 2, \dots, m$, se cumple que $x_{pj} \geq x_{qj}$.

Demostración: Probamos esto por inducción en j . El caso $j = 1$ se deduce directamente de la definición del algoritmo codicioso: tus amigos viajan el mayor tiempo posible el primer día antes de detenerse. Ahora deje que $j > 1$ y suponga que la afirmación es verdadera para todo $i < j$. Entonces $x_{qj} - x_{q_{j-1}} \leq d$ ya que S es un conjunto valido de puntos de acampe, y $x_{pj} - x_{p_{j-1}} \leq x_{qj} - x_{q_{j-1}}$ ya que $x_{p_{j-1}} \geq x_{q_{j-1}}$ por hipótesis de inducción. Combinando estas dos se da con la siguiente inecuación: $x_{qj} - x_{p_{j-1}} \leq d$.

Esto significa que tus amigos tienen la opción de ir de $x_{p_{j-1}}$ a x_{qj} en un día; y, por lo tanto, la ubicación x_{pj} en la que finalmente se detienen solo puede estar más lejos que x_{qj} . (Tenga en cuenta la similitud con la prueba correspondiente para el problema de programación de intervalos: aquí también el algoritmo codicioso se mantiene adelante porque, en cada paso, la elección realizada por la solución alternativa es una de sus opciones válidas).

La declaración implica en particular que $x_{q_m} \leq x_{p_m}$. Ahora, si $m < k$, entonces debemos tener $x_{p_m} < L - d$ (siendo L el punto de llegada), de lo contrario sus amigos nunca habrían tenido que detenerse en la ubicación $x_{p_{m+1}}$. Combinando estas dos desigualdades, hemos concluido que $x_{q_m} < L - d$; pero esto contradice la suposición de que S es un conjunto válido de puntos de parada.

En consecuencia, no podemos tener $m < k$, por lo que hemos demostrado que el algoritmo codicioso produce un conjunto válido de puntos de detención del tamaño mínimo posible.

The Security company.

Enunciado.

Sus amigos están comenzando una compañía de seguridad que necesita obtener licencias para n diferentes piezas de software criptográfico. Debido a las regulaciones, solo pueden obtener estas licencias a razón de una por mes como máximo.

Cada licencia se vende actualmente por un precio de \$100. Sin embargo, todos se están volviendo más caros de acuerdo con las curvas de crecimiento exponencial: en particular, el costo de la licencia j aumenta en un factor de $r_j > 1$ cada mes, donde r_j es un parámetro dado. Esto significa que si la licencia j se compra dentro de unos t meses, costará $100 \cdot r_j^t$. Asumiremos que todas las tasas de crecimiento de los precios son distintas; es decir, $r_i \neq r_j$ para licencias $i \neq j$ (a pesar de que comienzan con el mismo precio de \$100).

La pregunta es: dado que la empresa solo puede comprar como máximo una licencia por mes, ¿en qué orden debe comprar las licencias para que la cantidad total de dinero que gasta sea lo más pequeña posible?

Proporcione un algoritmo que tome las n tasas de crecimiento de precios r_1, r_2, \dots, r_n , y calcule un pedido en el que comprar las licencias para minimizar la cantidad total de dinero gastado. El tiempo de ejecución de su algoritmo debe ser polinomial en n .

Diseño de la solución.

Para comenzar, diremos que una solución cualquiera se verá representada por una secuencia $[R_{j_1}, \dots, R_{j_n}]$ de factores ordenados según un criterio, mediante el cual se sabe que licencias se pagarán en un mes $i \in [1, n]$.

Nuestra solución se basa en ordenar los factores R_j en forma decreciente, tal que:

$$\sum_{i=1}^n 100 * R_{j_i}^i$$

sea mínima. Intentemos probar que ordenar el r_i en orden decreciente de hecho siempre da la solución óptima. Cuando un algoritmo codicioso funciona para problemas como este, en el que ponemos un conjunto de cosas en un orden óptimo, hemos visto en el texto que a menudo es efectivo intentar probar la corrección utilizando un argumento de intercambio.

Para hacer esto aquí, supongamos que hay una solución óptima O que difiere de nuestra solución S . (En otras palabras, S consiste en las licencias ordenadas en orden decreciente). Entonces, esta solución óptima O debe contener una inversión, es decir, deben existir dos meses vecinos t y $t + 1$, de modo que la tasa de aumento de precio de la licencia comprada en el mes t (denotémosla por r_t) sea menor que la comprada en el mes $t + 1$ (de manera similar, usamos r_{t+1} para denotar esto). Es decir, tenemos $r_t < r_{t+1}$.

Afirmamos que al intercambiar estas dos compras, podemos mejorar estrictamente nuestra solución óptima, lo que contradice la suposición de que O era óptimo. Por lo tanto, si lo logramos, mostraremos con éxito que nuestro algoritmo es el correcto. Tenga en cuenta que si intercambiamos estas dos compras, el resto de las compras tienen un precio idéntico. En O , la cantidad pagada durante los dos meses involucrados en el intercambio es $100(r_t^t + r_{t+1}^{t+1})$. Por otro lado, si intercambiamos estas dos compras, pagaríamos $100(r_{t+1}^t + r_t^{t+1})$. Dado que la constante 100 es común a ambas expresiones, queremos mostrar que el segundo término es menor que el primero. Entonces queremos mostrar que

$$\begin{aligned}
r_{t+1}^t + r_t^{t+1} &\leq r_t^t + r_{t+1}^{t+1} \\
r_t^{t+1} - r_t^t &\leq r_{t+1}^{t+1} - r_t^{t+1} \\
r_t^t(r_t - 1) &\leq r_{t+1}^t(r_{t+1} - 1)
\end{aligned}$$

Pero esta última desigualdad es verdadera simplemente porque $r_i > 1$ para todo i y desde $r_t < r_{t+1}$.

Esto concluye la prueba de corrección. El tiempo de ejecución del algoritmo es $O(n \log n)$, ya que la clasificación lleva tanto tiempo y el resto (salida) es lineal. Entonces, el tiempo de ejecución general es $O(n \log n)$.

Nota: Es interesante notar que las cosas se vuelven mucho menos directas si variamos esta pregunta aunque sea un poco. Suponga que en lugar de comprar licencias cuyos precios aumentan, está tratando de vender equipos cuyo costo se está depreciando. El artículo i se deprecia a un factor de $r_t < 1$ por mes, a partir de \$100, por lo que si lo vende dentro de unos meses recibirá $100 \cdot r_t^t$. (En otras palabras, las tasas exponenciales ahora son menores que 1, en lugar de mayores que 1.) Si solo puede vender un artículo por mes, ¿cuál es el orden óptimo para venderlos? Aquí, resulta que hay casos en los que la solución óptima no pone las tasas en orden creciente o decreciente.

Las propiedades de los arboles de recubrimiento mínimo

Enunciado

Suponga que se le da un grafo G conectado, con costos de aristas que puede suponer que son todos distintos. G tiene n vértices y m aristas. Se especifica una arista e particular de G . Proporcione un algoritmo con tiempo de ejecución $O(m + n)$ para decidir si e está contenido en un árbol de expansión mínimo de G .

Diseño de la solución

Conocemos dos reglas mediante las cuales podemos concluir si una arista e pertenece a un árbol de expansión mínimo: la propiedad de corte dice que e está en cada árbol de expansión mínimo cuando es el cruce de aristas más barato de algunos establezca S en el complemento $V - S$; y la propiedad del ciclo dice que e no está en un árbol de expansión mínimo si es la arista más cara en algún ciclo C . Veamos si podemos hacer uso de estas dos reglas como parte de un algoritmo que resuelve este problema en tiempo lineal.

Tanto las propiedades de corte como las de ciclo están hablando esencialmente de cómo se relaciona con el conjunto de aristas que son más baratos que e . La propiedad de corte puede verse como una pregunta: ¿Existe algún conjunto $S \subseteq V$ de modo que para pasar de S a $V - S$ sin usar e , necesitemos usar una arista que sea más cara que e ? Y si pensamos en el ciclo C en el enunciado de la propiedad del ciclo, recorrer el "camino largo" alrededor de C (evitando e) puede verse como una ruta alternativa entre los extremos de e que solo usa aristas más baratas.

Poner estas dos observaciones juntas sugiere que deberíamos intentar proporcionar la siguiente afirmación.

Lema: La arista $e = (v, w)$ no pertenece a un árbol de expansión mínimo de G si y solo si v y w pueden unirse mediante una ruta que consta completamente de aristas que son más baratas que e .

Demuestra: Primero suponga que P es un camino $v-w$ que consta completamente de aristas más baratas que e . Si sumamos e a P , obtenemos un ciclo en el que e es el camino más caro. Por lo tanto, según la propiedad del ciclo, e no pertenece a un árbol de expansión mínimo de G .

Por otro lado, suponga que v y w no se pueden unir mediante un camino que consta completamente de aristas más baratas que e . Ahora identificaremos un conjunto S para el cual e es la arista más barata con un extremo en S y el otro en $V - S$; si podemos hacer esto, la propiedad de corte implicará que e pertenece a cada árbol de expansión mínimo. Nuestro conjunto S será el conjunto de todos los nodos a los que se puede acceder desde v utilizando una ruta que consta solo de aristas que son más baratas que e . Según nuestro supuesto, tenemos $w \in V - S$. Además, según la definición de S , no puede haber una arista $f = (x, y)$ que sea más barata que e , y para la cual un extremo x se encuentra en S y el otro y se encuentra en $V - S$. De hecho, si existiera tal arista f , entonces, dado que el nodo x es accesible desde v

utilizando solo aristas más baratos que e , el nodo y también sería accesible. Por lo tanto, e es la arista más barata con un extremo en S y el otro en $V - S$.

Dado este hecho, nuestro algoritmo ahora es simplemente el siguiente. Formamos un grafo G eliminando de G todos las aristas de peso mayor que c_e (así como eliminando e mismo). Luego usamos uno de los algoritmos de conectividad para determinar si hay una ruta de v a w en G . La declaración (4.41) dice que e pertenece a un árbol de expansión mínimo si y solo si no existe tal ruta. El tiempo de ejecución de este algoritmo es $O(m + n)$ para construir G , y $O(m + n)$ para probar una ruta de v a w .

División y Conquista.



En general, el estilo de division y conquista es muy simple comprendiendo tan solo los siguientes pasos:

- Dividir el problema en un numero de subproblemas que son instancias mas pequeñas que del problema inicial.
- Conquistar los subproblemas resolviéndolos recursivamente. Sin embargo, si los tamaños de los subproblemas son lo suficientemente pequeños, simplemente resuelva los subproblemas de manera directa.
- Combinar las soluciones de los sub problemas en una solucion que satisfaga al problema inicial.

Cuando los subproblemas son lo suficientemente grandes como para resolverse de forma recursiva, lo llamamos el caso recurrente. Una vez que los subproblemas se vuelven lo suficientemente pequeños como para que ya no recurramos, decimos que la recursión "toca fondo" y que hemos llegado al caso base. A veces, además de los subproblemas que son instancias más pequeñas del mismo problema, tenemos que resolver los subproblemas que no son exactamente lo mismo que el problema original. Consideraremos resolver tales subproblemas como parte del paso combinado. En este capítulo, veremos más algoritmos basados en divide-and-conquer. El primero resuelve el problema de la submatriz máxima: toma como entrada una matriz de números y determina la submatriz contigua cuyos valores tienen la mayor suma. Luego veremos dos algoritmos de divide y vencerás para multiplicar $n \times n$ matrices. Uno corre en n^3 tiempo, que no es mejor que el método directo de multiplicar matrices cuadradas. Pero el otro, el algoritmo de Strassen, se ejecuta en $n^{2.81}$, que supera el método directo de forma asintótica.

El algoritmo Mergesort.

La receta general para los problemas de división y conquista se puede resumir de la siguiente forma:

Divida la entrada en x piezas de igual tamaño; resuelva los x sub problemas en estas piezas por separado por recursividad; y luego combine los x resultados en una solución general, gastando solo tiempo lineal para la división inicial y la recombinación final.

En Mergesort se sigue esta receta al dividir el problema en dos partes de tamaños iguales, se intenta ordenar ambas partes, para luego combinar sus resultados.

En Mergesort, como en cualquier algoritmo que se ajuste a este estilo, también necesitamos un caso base para la recursión, que generalmente tiene un "fondo" en las entradas de algún tamaño constante. En el caso de Mergesort, asumiremos que una vez que la entrada se ha reducido al tamaño 2, detenemos la recursividad y clasificamos los dos elementos simplemente comparándolos entre sí.

Considere cualquier algoritmo que se ajuste al patrón, y deje que $T(n)$ denote su peor tiempo de ejecución en instancias de entrada de tamaño n . Suponiendo que n es par, el algoritmo pasa $O(n)$ tiempo para dividir la entrada en dos piezas de tamaño $n/2$ cada una; luego pasa el tiempo $T(n/2)$ para resolver cada uno (ya que $T(n/2)$ es el peor tiempo de ejecución para una entrada de tamaño $n/2$); y finalmente pasa $O(n)$ tiempo para combinar las soluciones de las dos llamadas recursivas. Así, el tiempo de ejecución $T(n)$ satisface la siguiente relación de recurrencia.

$$T(n) = \begin{cases} c & \text{si } n \leq 2 \\ 2T(n/2) + cn & \text{si } n > 2 \end{cases}$$

Para resolver esta ecuación de recurrencia, en general se apela al **Teorema del Maestro** que enunciaremos a continuación:

Sean $a \geq 1$ y $b \geq 1$ constantes, sea una función $f(n)$ y una función recursiva $T(n)$ definida sobre enteros no negativos, tal que:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Se puede decir que $T(n)$ tiene los siguientes límites asintóticos:

- Caso 1: Si $f(n) = O(n^{\log_b a - \epsilon})$, para una constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_b a})$.
- Caso 2: Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} * \log(n))$.
- Caso 3: Si $f(n) = \Omega(n^{\log_b a + \epsilon})$, para una constante $\epsilon > 0$, y si $a * f(\frac{n}{b}) \leq c * f(n)$ para alguna constante $c < 1$, y todo con un n suficientemente grande, entonces $T(n) = \Theta(f(n))$.

Nota: No siempre es posible aplicar el Método del Maestro, por lo que en esos casos solo restaría resolver la ecuación de recurrencia que se plantea.

En nuestro caso, esto se resume a que $a = 2$, $b = 2$ y $f(n) = n$. Analizamos el caso 2 damos con que $\Theta(n^{\log_2 2}) = \Theta(n) = f(n)$, por tanto, podemos concluir que el orden de ejecución del Algoritmo de Mergesort es $O(n \log(n))$

Veamos algunos ejemplos de uso del Teorema del Maestro

- $T(n) = 9T(n/3) + n$: En este caso, tenemos que $a = 9$, $b = 3$ y $f(n) = n$. Rapidamente decimos que $n^{\log_3 9} = n^2$. Esto ultimo lo calculamos para aproximar mejor cual sera el caso a usar. En este punto podemos ver que $f(n) = O(n^{2-\epsilon})$, es decir, $f(n)$ esta acotado superiormente por la función $O(n^{2-\epsilon})$ con un $\epsilon = 1$, lo que nos conduce al caso 1 y concluimos en que $T(n) = \Theta(n^2)$.
- $T(n) = T(2n/3) + 1$: En este caso, tenemos que $a = 1$, $b = 3/2$ y $f(n) = 1$. Rapidamente decimos que $n^{\log_{3/2} 1} = n^0 = 1$. En este caso podemos ver que $f(n) = \Theta(n^0) = \Theta(1)$, es decir, $f(n)$ esta acotado superior e inferiormente por la función $\Theta(1)$ lo que nos conduce al caso 2 y concluimos en que $T(n) = \Theta(\log(n))$.
- $T(n) = 3T(n/4) + n \log(n)$: En este caso, tenemos que $a = 3$, $b = 4$ y $f(n) = n \log(n)$. Rapidamente decimos que $n^{\log_4 3} = n^{\log_4 3} = n^{0.79}$. En este caso podemos ver que $f(n) = \Omega(n^{0.79+\epsilon}) = \Omega(n)$, es decir, $f(n)$ esta acotado inferiormente por la función $\Omega(n)$ con un $\epsilon = 0.21$. Ahora hay que analizar si $a * f(\frac{n}{b}) \leq c * f(n)$ para alguna constante $c < 1$,

$$a * f\left(\frac{n}{b}\right) \leq c * f(n)$$

$$3\left(\frac{n}{4} \log\left(\frac{n}{4}\right)\right) \leq cn \log(n)$$

Podemos proponer $c = \frac{3}{4}$

$$\frac{3}{4}n \log\left(\frac{n}{4}\right) \leq \frac{3}{4}n \log(n)$$

$$n \log\left(\frac{n}{4}\right) \leq n \log(n)$$

Concluimos en que $T(n) = \Theta(n \log(n))$.

Counting Inversions

Enunciado

Consideraremos un problema que surge en el análisis de rankings, que están cobrando importancia para una serie de aplicaciones actuales. Por ejemplo, varios sitios en la Web utilizan una técnica conocida como filtrado colaborativo, en la que intentan hacer coincidir sus preferencias (libros, películas, restaurantes) con las de otras personas en Internet. Una vez que el sitio web ha identificado a las personas con gustos "similares" a los tuyos, basándose en una comparación de cómo usted y ellos califican varias cosas, puede recomendar cosas nuevas que les hayan gustado a estas otras personas. Otra aplicación surge en las herramientas de metabúsqueda en la Web, que ejecutan la misma consulta en muchos buscadores diferentes y luego intentan sintetizar los resultados buscando similitudes y diferencias entre los distintos rankings que devuelven los buscadores.

Un tema central en aplicaciones como esta es el problema de comparar dos clasificaciones. Clasifica un conjunto de n películas y luego un sistema de filtrado colaborativo consulta su base de datos para buscar otras personas que tuvieran clasificaciones "similares". Pero, ¿cuál es una buena manera de medir numéricamente qué tan similares son las clasificaciones de dos personas? Claramente, una clasificación idéntica es muy similar y una clasificación completamente invertida es muy diferente; queremos algo que se interpola a través de la región media.

Consideraremos comparar su clasificación y la clasificación de un extraño del mismo conjunto de n películas. Un método natural sería etiquetar las películas del 1 al n según su clasificación, luego ordenar estas etiquetas según la clasificación del extraño y ver cuántos pares están "fuera de orden". Más concretamente, consideraremos el siguiente problema. Se nos da una secuencia de n números a_1, \dots, a_n ; asumiremos que todos los números son distintos. Queremos definir una medida que nos diga qué tan lejos está esta lista de estar en orden ascendente; el valor de la medida debe ser 0 si $a_1 < a_2 < \dots < a_n$, y debería aumentar a medida que los números se vuelven más mezclados.

Una forma natural de cuantificar esta noción es contando el número de inversiones. Decimos que dos índices $i < j$ forman una inversión si $a_i > a_j$, es decir, si los dos elementos a_i y a_j están "fuera de orden". Buscaremos determinar el número de inversiones en la secuencia a_1, \dots, a_n .

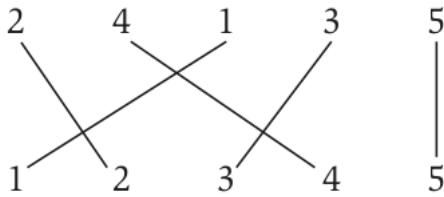


Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

Diseño de la solución

¿Cuál es el algoritmo más simple para contar inversiones? Claramente, podríamos mirar cada par de números (a_i, a_j) y determinar si constituyen una inversión; esto tomaría $O(n^2)$ tiempo.

Ahora mostramos cómo contar el número de inversiones mucho más rápidamente, en tiempo $O(n \log n)$. Tenga en cuenta que, dado que puede haber un número cuadrático de inversiones, dicho algoritmo debe poder calcular el número total sin siquiera mirar cada inversión individualmente. La idea básica es seguir la estrategia (†) definida anteriormente. Establecemos $m = n/2$ y dividimos la lista en dos partes a_1, \dots, a_m y a_{m+1}, \dots, a_n . Primero contamos el número de inversiones en cada una de estas dos mitades por separado. Luego contamos el número de inversiones (a_i, a_j) , donde los dos números pertenecen a mitades diferentes; el truco es que debemos hacer esta parte en $O(n)$ tiempo, si queremos aplicar. Tenga en cuenta que estas inversiones de la primera mitad / segunda mitad tienen una forma particularmente agradable: son precisamente los pares (a_i, a_j) , donde a_i está en la primera mitad, a_j está en la segunda mitad y $a_i > a_j$.

Para ayudar a contar el número de inversiones entre las dos mitades, haremos que el algoritmo también ordene recursivamente los números en las dos mitades. Hacer que el paso recursivo trabaje un poco más (ordenar y contar inversiones) facilitará la parte de "combinación" del algoritmo.

Entonces, la rutina crucial en este proceso es Merge-and-Count. Suponga que hemos ordenado recursivamente la primera y segunda mitades de la lista y contado las inversiones en cada una. Ahora tenemos dos listas ordenadas A y B , que contienen la primera y segunda mitades, respectivamente. Queremos producir una sola lista ordenada C a partir de su unión, al tiempo que contamos el número de pares (a, b) con $a \in A$, $b \in B$ y $a > b$. Según nuestro análisis anterior, esto es precisamente lo que necesitaremos para el paso de "combinación" que calcula el número de inversiones de la primera mitad y la segunda mitad.

Resulta que podremos hacer esto con el mismo estilo que usamos para la fusión. Nuestra rutina Merge-and-Count recorrerá las listas ordenadas A y B , eliminando elementos del frente y agregándolos a la lista ordenada L . En un paso dado, tenemos un puntero actual en cada lista, mostrando nuestra posición actual. Suponga que estos punteros se encuentran actualmente en los elementos a_i y b_j . En un paso, comparamos los elementos a_i y b_j a los que se apunta en cada lista, eliminamos el más pequeño de su lista y lo agregamos al final de la lista L .

Esto se encarga de fusionar. ¿Cómo contamos también el número de inversiones? Debido a que A y B están ordenados, en realidad es muy fácil realizar un seguimiento del número de inversiones que encontramos. Cada vez que el elemento a_i se agrega a L , no se encuentran nuevas inversiones, ya que a_i es más pequeña que todo lo que queda en la lista B , y está antes que todas. Por otro lado, si b_j se agrega a la lista L , entonces es más pequeño que todos los elementos restantes en A , y viene después de todos ellos, por lo que aumentamos nuestro recuento del número de inversiones por el número de elementos que quedan en A . Ésta es la idea crucial: en tiempo constante, hemos contabilizado un número potencialmente grande de inversiones.

```

Definimos el algoritmo MergeAndCount que recibe dos listas A y B:
Mantenemos un puntero en cada una de las listas puestas al inicio.
Mantenemos una variable C para el numero de inversiones.
Sea una lista vacia L.
Mientras ambas listas no esten vacias:
    Sean ai y bj los elementos apuntados en cada lista.
    Agregamos el mas pequeño de los dos a la lista de salida L.
    if bj es el elemento mas pequeño.
        Incrementamos el contador C por el numero de elementos remanentes en A.

    Movemos el puntero de la lista con el elemento mas pequeño que fue seleccionado.

    Una vez que una de las listas este vacia, agregamos el remanente de la otra a la lista L.
    Devolvemos la lista L y el contador C.

Sea una lista L de n elementos.
Ejecutamos el algoritmo SortAndCount que recibe L como parametro:
Si la lista L tiene un solo elemento:
    Entonces no hace falta inversiones y devolvemos 0.
Sino:
    Dividimos la lista L en A y B.
    (ra, A)=SortAndCount(A).
    (rb, B)=SortAndCount(B).
    (r, L)=MergeAndCount(A, B).
    Devolvemos ra + rb + r.

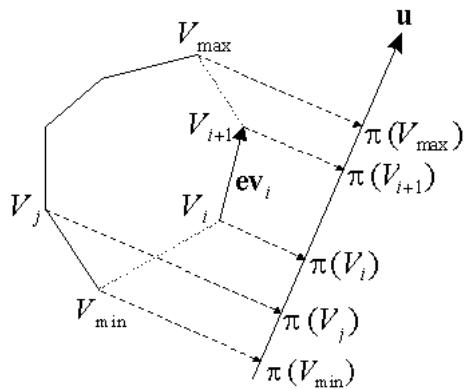
```

En definitiva, estamos ordenando y contando las inversiones necesarias. Esto es similar a Merge-Sort y cuenta con el mismo orden de ejecución, o sea, $O(n \log(n))$.

Puntos extremos de un polígono convexo.

Enunciado.

Entonces, sea P un polígono definido a través de un conjunto de vértices ordenados $V = v_0, v_1, \dots, v_n$. Sea e_i la i -ésima arista entre los vértices v_i y v_{i+1} , y sea $ev_i = v_{i+1} - v_i$ el i -ésimo vector-arista. Se desea encontrar los vértices extremos máximos y mínimos de P en la dirección u . En la figura se puede ver expresado como los vértices que proyectados sobre la recta u de acuerdo a una función $\pi : R^2 \rightarrow R$ adquieren un orden. Esto es lo que define cuando un vértice está por encima de otro, respecto a una dirección u y cuando un vértice es un extremo. (básicamente, cuando está por encima de todos los demás vértices, con respecto a la dirección u).



Diseño de la solución

Método para la comparación entre vértices: Sea un polígono convexo y por tanto monótono P con un conjunto de vértices $V = \{v_0, v_1, \dots, v_n\}$ y sea una recta L con dirección en u . Decimos que $v_i > v_j$ con relación a u , si $\pi(v_i) > \pi(v_j)$. Si u es un versor (y si no lo es, podemos construirlo dividiendo cada uno de sus componentes por el modulo del vector), podemos proponer que:

$$\begin{aligned}\pi(v_i) &> \pi(v_j) \\ u * v_i &> u * v_j \\ u * v_i - u * v_j &> 0 \\ u * (v_i - v_j) &> 0\end{aligned}$$

Pensemos por un momento que la dirección del vector u coincide con la del eje x del plano cartesiano. Desde este punto de vista, lo que obtendremos no es mas los vértices de P cuyas componentes en x son máximas y mínimas, idénticamente a lo que se obtuvo en la solución inicial.

A continuación, se exponen dos conceptos que marcaran la diferencia entre la solución propuesta y otras posibles soluciones.

Polígonos monótonos: Se dice que un polígono P es monótono respecto a una recta L con dirección u , si la misma corta a P en solo dos puntos, generando dos segmentos monótonos, uno creciente y otro decreciente en relación con la dirección u . Se puede decir que todo polígono convexo es monótono.

*Segmentos monótonos ascendentes y descendentes: Sea un polígono convexo y por tanto monótono P con un conjunto de vértices $V = \{v_0, v_1, \dots, v_n\}$ y sea una recta L con dirección en u . Si resulta que $u * (v_{i+1} - v_i) > 0$, quiere decir que la arista e_i es ascendente con respecto a u , o lo que es equivalente, ev_i es un vector ascendente con respecto a u . En caso contrario, se considera al segmento como descendente con respecto a u .*

El hecho de que los polígonos convexos sean monótonos, y que se pueda definir con cierta facilidad que aristas del mismo son ascendentes o descendentes respecto a una dirección u es lo que nos va permitir definir extremos locales como globales (ya que de existir mas de un extremo local con respecto a u representaría una discontinuidad en la monotonía del polígono, negando así, su condición de convexo, lo cual no puede ser).

Dado que el conjunto de puntos que definen al polígono P para este problema, es un conjunto ordenado (supondremos que están ordenados en el sentido antihorario) de vértices $V = \{v_0, v_1, \dots, v_n\}$, entonces podemos encontrar el vértice extremo máximo (y similarmente, el mínimo) relativo a u más rápidamente, con una búsqueda binaria.

```
Sea P un polígono expresado mediante un arreglo de vectores de n
posiciones.
Sea R, un rectángulo determinado por los vértices que lo componen.
Sean XMAX, XMIN, YMAX, YMIN variables.
Sea A una variable inicializada en 0.
```

```

Sea B una variable inicializada en n-1
Sea u el vector (1,0)
Sea un el vector (-1,0)
Sea w el vector (0,1)
Sea wn el vector (0,-1)
Sea ve el vector (0,0)

// Todo producto entre vectores es un producto escalar

Sea la funcion ES_UN_EXTREMO aplicada sobre un indice C y un versor
U:
    Devolver P(C)*U >= P(C-1)*U y P(C)*U >= P(C+1)*U.

Sea la funcion ASCENDENTE_RESPECTO_A_U aplicada sobre un indice A y
un versor U:
    Devolver (P(A+1) - P(A)) * U > 0.

Sea la funcion ESTA POR ENCIMA aplicada sobre los indices A, C y un
versor U:
    Devolver P(A)*U >= P(C)*U

Sea la funcion recursiva OROURKE aplicada sobre P con indices A, B,
y un versor U:
    Si A + 1 >= B:
        Emitir "error poligono demasiado pequenio".

    Si ES_UN_EXTREMO(A, U):
        Devolvemos P(A).

    Si ES_UN_EXTREMO(B, U):
        Devolvemos P(B).

    Sea C una variable igual a (A+B)/2.
    Si ES_UN_EXTREMO(C, U):
        Devolvemos P(C).

    Si ASCENDENTE_RESPECTO_A_U(A, U):
        Si !ASCENDENTE_RESPECTO_A_U(C, U):
            Devolvemos OROURKE(A,C,U).
        Sino:
            Si ESTA_POR_ENCIMA(C, A, U):
                Devolvemos OROURKE(C,B,U).
            Sino:
                Devolvemos OROURKE(A,C,U).
        Sino:
            Si ASCENDENTE_RESPECTO_A_U(C, U):
                Devolvemos OROURKE(C,B,U).
            Sino:
                Si ESTA_POR_ENCIMA(A, C, U):
                    Devolvemos OROURKE(C,B,U).
                Sino:
                    Devolvemos OROURKE(A,C,U).

VE = OROURKE(A,B,U).
Asignamos VE.x a XMAX.

VE = OROURKE(A,B,UN).
Asignamos VE.x a XMIN.

VE = OROURKE(A,B,W).
Asignamos VE.y a YMAX.

VE = OROURKE(A,B,WN).
Asignamos VE.y a YMIN.

```

```

se agrega el vertice (XMAX, YMIN) a R.
se agrega el vertice (XMIN, YMIN) a R.
se agrega el vertice (XMIN, YMAX) a R.
se agrega el vertice (XMAX, YMAX) a R.

```

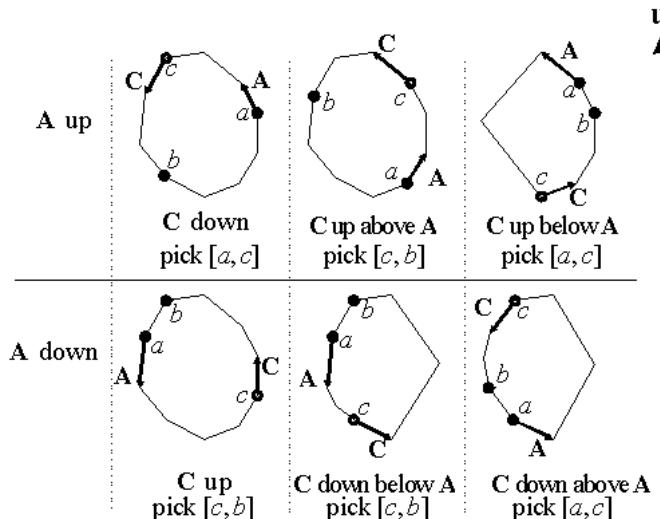
Devolvemos R.

Supongamos que uno de los extremos se encuentra entre los vértices v_a y v_b . Lo primero que se verifica en la función principal OROURKE es si los vértices v_a y v_b no son en sí, extremos locales. En tal caso, se finaliza con la ejecución.

En cualquier otro caso, se procede a elegir un vértice v_c intermedio entre v_a y v_b . Si es un extremo local, entonces es también un extremo global y se habrá terminado la búsqueda, sino, se debe restringir la búsqueda a los vértices que se encuentran entre v_a y v_c o bien entre v_c y v_b .

Debido a que el polígono es convexo, podemos determinar fácilmente cuál cadena (conjunto de vértices secuenciales según el orden en que se definieron.) elegir al comparar las posiciones relativas y las direcciones respecto de u de los vectores ev_a y ev_c .

Se pueden presentar seis casos, tres con ev_a ascendente y tres con ev_a descendente, como se muestra en la figura.



De arriba hacia abajo y de izquierda a derecha: La primera figura muestre el caso en que ev_c es descendente. El segundo es el caso en que $v_c > v_a$ respecto de u . En el tercer caso, $v_a > v_c$ respecto de u . En el cuarto caso, ev_c es el ascendente. En el quinto caso $v_a > v_c$. En el sexto caso $v_c > v_a$.

Pero nuestra misión no culmina sino hasta encontrar los cuatro vértices extremos. Para lograr esto, se decidió repetir el proceso anterior, pero con las direcciones $-u, w, -w$ \footnote{Si bien, existen varios métodos para encontrar el vértice extremo mínimo respecto de u , en nuestra solución se decidió por invertir la dirección de u y repetir la misma secuencia de pasos buscando el máximo extremo.}, con w ortogonal a u .

Como detalle de implementación, se tomaran como direcciones u y w , a los vectores $(1, 0)$ y $(0, 1)$, es decir, versores que coinciden con la dirección de los ejes x e y respectivamente. Esto no solo facilitara los cálculos y la construcción de la solución final, sino que nos permitirá hacer una comparación entre los resultados obtenidos mediante la presente solución y la mencionada en la sección anterior.

Analisis de la solución

La solución optimizada hace uso de una función recursiva para encontrar cada uno de los extremos. Cada vez que se llama a la función, se ejecutan los siguientes pasos:

- En primer lugar, la solución intenta dar con un extremo global. Si lo encuentra el algoritmo termina, pero sino, debe analizar en el peor de los casos, seis opciones

posibles para poder continuar.

- Tras determinar cual es la opción mas adecuada a su situación, toma la mitad de los vértices que consideraba hasta el momento, para profundizar su búsqueda sobre esa porción de los datos.
- En tanto el polígono sea convexo, la solución propuesta va a encontrar a los extremos requeridos y finalizara con normalidad.

A partir de esto, se puede afirmar que la complejidad temporal de la función recursiva se comporta de acuerdo a la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(n/2) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Aplicamos entonces el método del maestro para poder resolver esta ecuación de recurrencia.

Teorema del maestro: Sean $a \geq 1$ y $b \geq 1$ constantes, sea una función $f(n)$ y una función recursiva $T(n)$ definida sobre enteros no negativos, tal que:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Se puede decir que $T(n)$ tiene los siguientes límites asintóticos:

- Caso 1: Si $f(n) = O(n^{\log_b a - \epsilon})$, para una constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_b a})$.
- Caso 2: Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} * \log(n))$.
- Caso 3: Si $f(n) = \Omega(n^{\log_b a + \epsilon})$, para una constante $\epsilon > 0$, y si $a * f\left(\frac{n}{b}\right) \leq c * f(n)$ para alguna constante $c < 1$, y todo con un n suficientemente grande, entonces $T(n) = \Theta(f(n))$.

En nuestro caso $a = 1$, $b = 2$ y $f(n) = \Theta(1)$. Tenemos que $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$, entonces $\Theta(1) = f(n)$, y por tanto, podemos aplicar el caso 2, obteniendo el límite asintótico $T(n) = \Theta(\log(n))$.

Como la función recursiva se ejecuta cuatro veces, la complejidad de la solución optimizada es de $O(4\log(n)) = O(\log(n))$

La mediana de dos conjuntos de elementos separados.

Enunciado.

Supongamos que tenemos dos conjuntos ordenados de números sin repetir a los cuales, llamaremos A y B . Estos conjuntos poseen n elementos y lo que intentamos saber es la mediana de la unión $A \cup B$.

Diseño de la solución

Una solución por fuerza bruta es ir leyendo cada uno de los elementos de los conjuntos, hasta encontrar el n -ésimo elemento. Esto es $O(n)$. Pero existe una solución mejor.

El hecho de que ambos conjuntos están ordenados es una característica que debemos saber explotar al máximo. Es así que podríamos tomar un posible candidato a mediana como lo es el k -ésimo elemento de cada conjunto (siendo $k = n/2$). Claro que aun no podemos confirmar que estos sean la mediana, pero lo que si podemos hacer es descartar aquellos elementos que no pueden ser.

Si $A[k] < B[k]$ podemos estar seguros de que la mediana está entre $A[k]$ y $B[k]$. Esto es porque $A[k]$ es mayor a los primeros k elementos de A y $B[k]$ es el menor de los últimos k elementos de B , lo que descarta la posibilidad de que estos sean el n -ésimo elementos más pequeño que buscamos.

Lo que resta es continuar nuestra búsqueda de forma recursiva. Dejamos a continuación el pseudocódigo de la solución

```

Definimos la función Mediana que recibe un valor n, un puntero a y
un puntero b:
    k = techo(n/2)
    Si n==1:
        Devolvemos min(A[a+k], B[b=k])
    Si A[a+k] < B[b+k]:
        Devolvemos Mediana(k, a+piso(n/2), b)
    Sino:
        Devolvemos Mediana(k, a, b+piso(n/2))

```

Analisis de la solución

La solución divide a los datos de entrada en dos partes y solo trabaja sobre uno de ellos, lo que corresponde a la siguiente ecuación de recurrencia.

$$T(n) = T(n/2) + c$$

Mediante el teorema del maestro, podemos decir que el algoritmo presentado tiene una complejidad de $O(\log(n))$

Finding the Closest Pair of Points.

Enunciado.

El problema que consideramos es muy simple de enunciar: dados n puntos en el plano, encuentra el par más cercano.

El problema fue considerado por M. I. Shamos y D. Hoey a principios de la década de 1970, como parte de su proyecto para elaborar algoritmos eficientes para primitivas computacionales básicas en geometría. Estos algoritmos formaron las bases del campo incipiente de la geometría computacional, y han encontrado su camino en áreas como gráficos, visión por computadora, sistemas de información geográfica y modelado molecular. Y aunque el problema del par más cercano es uno de los problemas algorítmicos más naturales en geometría, es sorprendentemente difícil encontrar un algoritmo eficiente para él. Está claro de inmediato que hay una solución $O(n^2)$: calcule la distancia entre cada par de puntos y tome el mínimo, por lo que Shamos y Hoey preguntaron si se podía encontrar un algoritmo asintóticamente más rápido que el cuadrático. Pasaron bastante tiempo antes de que resolvieran esta pregunta, y el algoritmo $O(n \log n)$ que damos a continuación es esencialmente el que descubrieron. De hecho, cuando volvamos a este problema en Algoritmos Randomizados, veremos que es posible mejorar aún más el tiempo de ejecución a $O(n)$ utilizando la aleatorización.

Diseño de la solución

Comenzamos con un poco de notación. Denotemos el conjunto de puntos por $P = p_1, \dots, p_n$, donde p_i tiene coordenadas (x_i, y_i) ; y para dos puntos $p_i, p_j \in P$, usamos $d(p_i, p_j)$ para denotar la distancia euclídea estándar entre ellos. Nuestro objetivo es encontrar un par de puntos p_i, p_j que minimicen $d(p_i, p_j)$.

Configurando la recursividad

Primero vamos a sacar algunas cosas fáciles del camino. Será muy útil si cada llamada recursiva, en un conjunto $P' \subseteq P$, comienza con dos listas: una lista P'_x en la que todos los puntos en P' se han ordenado aumentando la coordenada x y una lista P'_y en el que todos los puntos en P' se han ordenado aumentando la coordenada y. Podemos asegurarnos de que esto siga siendo cierto en todo el algoritmo de la siguiente manera.

Primero, antes de que comience cualquiera de las recursiones, clasificamos todos los puntos en P por coordenadas x y nuevamente por coordenadas y, produciendo listas P_x y P_y . Adjunto a cada entrada en cada lista hay un registro de la posición de ese punto en ambas listas.

El primer nivel de recursión funcionará de la siguiente manera, y todos los niveles posteriores funcionarán de manera completamente análoga. Definimos Q como el conjunto de puntos en las primeras $[n / 2]$ posiciones de la lista Px (la "mitad izquierda") y R como el conjunto de puntos en las posiciones finales $[n / 2]$ de la lista Px (la "mitad derecha"). Con un solo paso a través de cada uno de Px y Py, en el tiempo O (n), podemos crear las siguientes cuatro listas: Qx, que consiste en los puntos en Q ordenados por coordenadas x aumentadas; Qy, que consiste en los puntos en Q ordenados aumentando la coordenada y; y listas análogas Rx y Ry. Para cada entrada de cada una de estas listas, como antes, registramos la posición del punto en ambas listas a las que pertenece.

Ahora determinamos recursivamente un par de puntos más cercano en Q (con acceso a las listas Qx y Qy). Suponga que q_0^* y q_1^* se devuelven (correctamente) como un par de puntos más cercano en Q. De manera similar, determinamos un par de puntos más cercano en R, obteniendo r_0^* y r_1^* .

Combinando las soluciones.

La maquinaria general de divide y vencerás nos ha llevado hasta aquí, sin que realmente hayamos profundizado en la estructura del problema del par más cercano. Pero aún nos deja con el problema que vimos que se avecinaba originalmente: ¿Cómo usamos las soluciones a los dos subproblemas como parte de una operación de "combinación" de tiempo lineal?

Sea δ el mínimo de $d(q_0^*, q_1^*)$ y $d(r_0^*, r_1^*)$. La verdadera pregunta es: ¿Hay puntos $q \in Q$ y $r \in R$ para los cuales $d(q, r) < \delta$? Si no, entonces ya hemos encontrado el par más cercano en una de nuestras llamadas recursivas. Pero si los hay, entonces los más cercanos q y r forman el par más cercano en P.

Deje que x^* denote la coordenada x del punto más a la derecha en Q, y deje que L denote la línea vertical descrita por la ecuación $x = x^*$. Esta línea L "separa" Q de R. Aquí hay un hecho simple.

Lema: Si existe $q \in Q$ y $r \in R$ para las cuales $d(q, r) < \delta$, entonces cada una de q y r se encuentra dentro de una distancia δ de L.

Entonces, si queremos encontrar un q y r cercano, podemos restringir nuestra búsqueda a la banda estrecha que consiste solo de puntos en P dentro de δ de L. Deje que $S \subseteq P$ denote este conjunto, y deje que Sy denote la lista que consiste en los puntos en S ordenado por el aumento de la coordenada y. Con un solo paso por la lista Py, podemos construir Sy en O (n) tiempo.

Lema: Existen $q \in Q$ y $r \in R$ para las cuales $d(q, r) < \delta$ si y solo si existen $s, s' \in S$ para las cuales $d(s, s') < \delta$.

Si $s, s' \in S$ tienen la propiedad de que $d(s, s') < \delta$, entonces s y s' están dentro de 15 posiciones entre sí en la lista ordenada Sy.

Notamos que el valor de 15 se puede reducir; pero para nuestros propósitos en este momento, lo importante es que es una constante absoluta.

En vista del lema anterior, podemos concluir el algoritmo de la siguiente manera.

Hacemos una pasada a través de Sy, y para cada $s \in S$, calculamos su distancia a cada uno de los siguientes 15 puntos en Sy. El enunciado implica que al hacerlo, habremos calculado la distancia de cada par de puntos en S (si los hay) que estén a una distancia menor que δ entre sí. Una vez hecho esto, podemos comparar la distancia más pequeña con δ , y podemos informar una de dos cosas: (i) el par de puntos más cercano en S, si su distancia es menor que δ ; o (ii) la conclusión (correcta) de que no hay pares de puntos en S dentro de δ entre sí. En el caso (i), este par es el par más cercano en P; en el caso (ii), el par más cercano encontrado por nuestras llamadas recursivas es el par más cercano en P. Tenga en cuenta la semejanza entre este procedimiento y el algoritmo que rechazamos al principio, que intentó hacer pasar una P a través de la coordenada y. La razón por la que este enfoque funciona ahora se debe al conocimiento adicional (el valor de δ) que hemos obtenido de las llamadas recursivas, y la estructura especial del conjunto S.

Esto concluye la descripción de la parte "combinada" del algoritmo, ya que en (5.9) ahora hemos determinado si la distancia mínima entre un punto en Q y un punto en R es menor que δ , y si es así, hemos encontrado el más cercano tal par.

The maximum-subarray problem.

Enunciado.

Suponga que se le ofrece la oportunidad de invertir en Volatile Chemical Corporation. Al igual que los productos químicos que produce la compañía, el precio de las acciones de Volatile Chemical Corporation es bastante volátil. Se le permite comprar una unidad de acciones solo una vez y luego venderla en una fecha posterior, comprando y vendiendo después del cierre de la operación del día. Para compensar esta restricción, se le permite saber cuál será el precio de la acción en el futuro. Su objetivo es maximizar sus ganancias. Puede comprar las acciones en cualquier momento, a partir del día 0, cuando el precio es de \$ 100 por acción. Por supuesto, usted querría "comprar bajo, vender alto" —comprar al precio más bajo posible y luego vender al precio más alto posible— para maximizar sus ganancias. Desafortunadamente, es posible que no pueda comprar al precio más bajo y luego vender al precio más alto dentro de un período determinado.

Puede pensar que siempre puede maximizar las ganancias comprando al precio más bajo y vendiendo al precio más alto. Si esta estrategia siempre funcionó, entonces sería fácil determinar cómo maximizar las ganancias: encontrar los precios más altos y más bajos, y luego trabajar izquierda desde el precio más alto para encontrar el precio anterior más bajo, trabajar directamente desde el precio más bajo para encontrar el precio más alto más tarde y tomar el par con la mayor diferencia. La Figura 4.2 muestra un contrajemplo simple, que demuestra que el beneficio máximo a veces no se obtiene comprando al precio más bajo ni vendiendo al precio más alto.

Para diseñar un algoritmo con un $O(n^2)$ tiempo de ejecución, veremos la entrada de una manera ligeramente diferente. Queremos encontrar una secuencia de días durante los cuales el cambio neto desde el primer día hasta el último es máximo. En lugar de mirar los precios diarios, consideraremos el cambio diario en el precio, donde el cambio en el día i es la diferencia entre los precios después del día $i - 1$ y después del día i . Si tratamos esta fila como un arreglo A, ahora queremos encontrar el subconjunto contiguo no vacío de A cuyos valores tienen la suma más grande. Llamamos a este subconjunto contiguo el subconjunto máximo.

A primera vista, esta transformación no ayuda. Todavía tenemos que verificar $O(n^2)$ sub arreglos por un período de n días. Entonces, busquemos una solución más eficiente para el problema de sub arreglos máximos. Al hacerlo, generalmente hablaremos de "una" submatriz máxima en lugar de "la" submatriz máxima, ya que podría haber más de una submatriz que logre la suma máxima. El problema de la submatriz máxima es interesante solo cuando la matriz contiene algunos números negativos. Si todas las entradas de la matriz no fueran negativas, entonces el problema de la submatriz máxima no presentaría ningún desafío, ya que toda la matriz daría la mayor suma.

Diseño de la solución.

Pensemos cómo podríamos resolver el problema de sub arreglo máximos utilizando la técnica de dividir y conquistar. Supongamos que queremos encontrar un sub arreglo máximo del sub arreglo $A[i, \dots, j]$. Dividir y conquistar sugiere que dividamos el subconjunto en dos subconjuntos del mismo tamaño posible. Es decir, encontramos el punto medio, digamos mid, del subconjunto, y consideramos los subconjuntos $A[low \dots mid]$, $A[mid + 1 \dots high]$. Entonces, el sub arreglo máximo $A[i \dots j]$ puede encontrarse

- Enteramente en el subconjunto $A[low \dots mid]$, entonces $low \leq i \leq j \leq mid$.
- Enteramente en el subconjunto $A[mid + 1 \dots high]$, entonces $mid \leq i \leq j \leq high$.
- Cruzando el punto medio o mid, entonces $low \leq i \leq mid \leq j \leq high$

A continuación damos a conocer el pseudocódigo para encontrar el sub arreglo máximo.

```
Sea un arreglo A.  
Sea una variable L inicializada con 0.  
Sea una variable H inicializada con n.  
Sea el algoritmo lineal MAX-CROSS-SUBARREGLO(A, L, M, H):  
    LEFT-SUM = -INFINITO.
```

```

SUM = 0.
Por cada valor i entre M y L:
    SUM += A[i].
    Si SUM > LEFT-SUM:
        LEFT-SUM = SUM.
        MAX-LEFT = i.

RIGHT-SUM = -INFINITO.
SUM = 0.
Por cada valor j entre M+1 y H:
    SUM += A[j]
    Si SUM > RIGHT-SUM:
        RIGHT-SUM = SUM.
        MAX-RIGHT = j.

Devolver (MAX-LEFT, MAX-RIGHT, LEFT-SUM + RIGHT-SUM).

Sea el algoritmo recursivo MAX-SUBREGLO(A, L, H):
Si H == L:
    Devolver (L, H, A[L]). 
Sino:
    M = (H + L) / 2.
    (LEFT-L, LEFT-H, LEFT-SUM) = MAX-SUBREGLO(A, L, M).
    (RIGHT-L, RIGHT-H, RIGHT-SUM) = MAX-SUBREGLO(A, M+1, H).
    (CROSS-L, CROSS-H, CROSS-SUM) = MAX-CROSS-
SUBREGLO(A,L,M,H).
    Si LEFT-SUM >= RIGHT-SUM y LEFT-SUM >= CROSS-SUM:
        Devolver (LEFT-L, LEFT-H, LEFT-SUM)
    Sino Si (RIGHT-SUM >= LEFT-SUM y RIGHT-SUM >= CROSS-SUM):
        Devolver (RIGHT-L, RIGHT-H, RIGHT-SUM)
    Sino:
        Devolver (CROSS-L, CROSS-H, CROSS-SUM)

```

Arreglos unimodales.

Enunciado

Suponga que se le da una matriz A con n entradas, y cada entrada contiene un número distinto. Se le dice que la secuencia de valores $A[1], A[2], \dots, A[n]$ es unimodal: para algún índice p entre 1 y n, los valores en las entradas de la matriz aumentan hasta la posición p en A y luego disminuyen el resto del camino hasta la posición n. (Entonces, si dibujara una gráfica con la posición de la matriz j en el eje x y el valor de la entrada $A[j]$ en el eje y, los puntos graficados aumentarían hasta el valor p en el eje x, donde alcanzar su máximo, y luego caer desde allí.)

Le gustaría encontrar la “entrada pico” p sin tener que leer toda la matriz; de hecho, leyendo la menor cantidad posible de entradas de A. Muestre cómo encontrar la entrada p leyendo como máximo O ($\log n$) entradas de A.

Diseño de la solución

Podríamos sondear el punto medio de la matriz e intentar determinar si la “entrada máxima” p se encuentra antes o después de este punto medio. Entonces supongamos que miramos el valor $A[n/2]$. A partir de este valor solo, no podemos decir si p se encuentra antes o después de $n/2$, ya que necesitamos saber si la entrada $n/2$ está asentada en una “pendiente ascendente” o en una “pendiente descendente”. Así que también miramos los valores $A[n/2 - 1]$ y $A[n/2 + 1]$. Ahora hay tres posibilidades.

- Si $A[n/2 - 1] < A[n/2] < A[n/2 + 1]$ entonces, el indice p se debe encontrar en la mitad derecha del arreglo.
- Si $A[n/2 - 1] > A[n/2] > A[n/2 + 1]$ entonces, el indice p se debe encontrar en la mitad izquierda del arreglo.
- Si $A[n/2 - 1] < A[n/2] > A[n/2 + 1]$ entonces, el indice p es $n/2$.

Este algoritmo tiene un orden de ejecución $O(\log(n))$.

Invertir en acciones

Enunciado

Está consultando para una pequeña empresa de inversión con uso intensivo de computación, y tienen el siguiente tipo de problema que quieren resolver una y otra vez. Un ejemplo típico del problema es el siguiente. Están haciendo una simulación en la que miran n días consecutivos de una acción determinada, en algún momento del pasado. Numeremos los días $i = 1, 2, \dots, n$; para cada día i , tienen un precio $p(i)$ por acción para las acciones de ese día. (Asumiremos, por simplicidad, que el precio se fijó durante cada día). Supongamos que durante este período de tiempo, quisieran comprar 1,000 acciones algún día y vender todas estas acciones algún día (posterior). Quieren saber: cuándo deberían haber comprado y cuándo no hubo forma de ganar dinero durante los n días, debería informar esto en su lugar).

Por ejemplo, suponga que $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Entonces debería devolver “compre en 2, venda en 3” (comprar el día 2 y vender el día 3 significa que habrían ganado

4 por acción, el máximo posible para ese período). Claramente, hay un algoritmo simple que lleva tiempo $O(n^2)$

: prueba todos los pares posibles de días de compra/venta y ve cuál les genera más dinero. Sus amigos inversionistas $O(n \log n)$.

Diseño de la solución

Ahora, sea S el conjunto de los días $1, \dots, n/2$, y S' el conjunto de días $n/2 + 1, \dots, n$. Nuestro algoritmo de divide y vencerás se basará en la siguiente observación: o existe una solución óptima en la que los inversores mantienen las acciones al final del día $n/2$, o no la hay. Ahora bien, si no la hay, entonces la solución óptima es la mejor de las soluciones óptimas en los conjuntos S y S' . Si hay una solución óptima en la que mantienen el stock al final del día $n/2$, entonces el valor de esta solución es $p(j) - p(i)$ donde $i \in S$ y $j \in S'$. Pero este valor se maximiza simplemente eligiendo $i \in S$ que minimiza $p(i)$ y eligiendo $j \in S'$ que maximiza $p(j)$.

Por lo tanto, nuestro algoritmo es tomar la mejor de las siguientes tres posibles soluciones.

- La solución optima esta en S .
- La solución optima esta en S' .
- La solución optima es $p(j) - p(i)$ donde $i \in S$ y $j \in S'$

Observamos que este no es el mejor tiempo de ejecución que se puede lograr para este problema. De hecho, uno puede encontrar el par de días óptimo en $O(n)$ tiempo usando la programación dinámica.

Programacion Dinamica.



La programación dinámica, como el método de divide y vencerás, resuelve problemas combinando las soluciones a subproblemas. ("Programación" en este contexto se refiere a un método tabular, no a escribir código de computadora). Los algoritmos de dividir y conquistar dividen el problema en subproblemas disjuntos, resuelven los subproblemas de forma recursiva y luego combine sus soluciones para resolver el problema original. Por el contrario, la programación dinámica se aplica cuando los subproblemas se superponen, es decir, cuando los subproblemas comparten subproblemas. En este contexto, un algoritmo de divide y vencerás hace más trabajo del necesario, solucionando repetidamente los subproblemas comunes. Un algoritmo de programación dinámica resuelve cada subproblema una sola vez y luego guarda su respuesta en una tabla, evitando así el trabajo de volver a calcular la respuesta cada vez que resuelve cada subproblema.

En general, se usa programacion dinamica cuando se encuentra frente a problema de optimizacion, donde existen multiples soluciones con un determinado valor y uno desea encontrar el maximo o el minimo de esos valores.

Elementos de la Programacion Dinamica.

Es posible que aún se pregunte cuándo se aplica el método. Desde una perspectiva de ingeniería, ¿cuándo debemos buscar una solución de programación dinámica para un problema? En esta sección, examinamos los dos ingredientes clave que debe tener un problema de optimización para que se aplique la programación dinámica: subestructura óptima y subproblemas superpuestos. También revisamos y discutimos más a fondo cómo la memorización podría ayudarnos a aprovechar la propiedad de subproblemas superpuestos en un enfoque recursivo de arriba hacia abajo.

Subestructura optima.

El primer paso para resolver un problema de optimización mediante programación dinámica es caracterizar la estructura de una solución óptima. Recuerde que un problema exhibe una subestructura óptima si una solución óptima al problema contiene soluciones óptimas a subproblemas. Cada vez que un problema exhibe una subestructura óptima, tenemos una buena pista de que la programación dinámica podría aplicarse. En la programación dinámica, creamos una solución óptima al problema desde soluciones óptimas hasta subproblemas. En consecuencia, debemos tener cuidado para asegurarnos de que el rango de subproblemas que consideramos incluya los utilizados en una solución óptima.

Te encontrarás siguiendo un patrón común para descubrir una subestructura óptima:

1. Usted muestra que una solución al problema consiste en hacer una elección, como elegir un corte inicial en una barra o elegir un índice en el que dividir la cadena de la matriz. Hacer esta elección deja uno o más subproblemas por resolver.
2. Supone que para un problema dado, se le da la opción que conduce a una solución óptima. Todavía no te preocupas por cómo determinar esta elección. Simplemente asumes que te lo han dado.

3. Dada esta opción, usted determina qué subproblemas se producen y cómo caracterizar mejor el espacio resultante de subproblemas.
4. Usted muestra que las soluciones a los subproblemas utilizados dentro de una solución óptima al problema deben ser óptimas utilizando una técnica de "cortar y pegar". Lo hace suponiendo que cada una de las soluciones de subproblemas no es óptima y luego deriva una contradicción. En particular, al "cortar" la solución no óptima a cada subproblema y "pegar" la solución óptima, demuestra que puede obtener una mejor solución al problema original, lo que contradice su suposición de que ya tenía una solución óptima. Si una solución óptima da lugar a más de un subproblema, generalmente son tan similares que puede modificar el argumento de cortar y pegar para que uno lo aplique a los demás con poco esfuerzo.

Para caracterizar el espacio de los subproblemas, una buena regla general dice tratar de mantener el espacio lo más simple posible y luego expandirlo según sea necesario. Por ejemplo, el espacio de subproblemas que consideramos para el problema del corte de varillas contenía los problemas de cortar de manera óptima una varilla de longitud i para cada tamaño i . Este espacio de subproblemas funcionó bien y no tuvimos necesidad de probar un espacio más general de subproblemas.

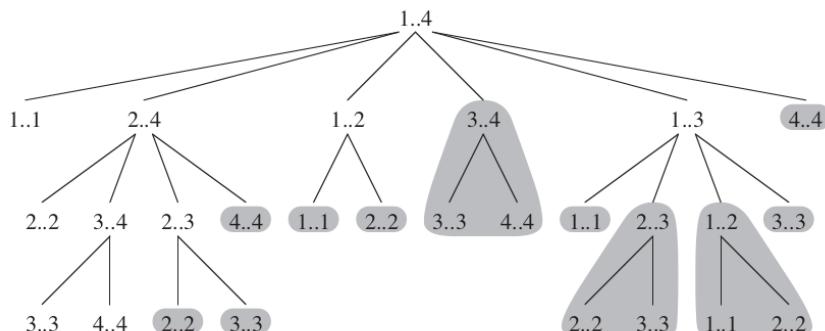
Por el contrario, supongamos que hemos tratado de restringir nuestro espacio de subproblema para la multiplicación de la cadena de matriz a productos de matriz de la forma A_1, A_2, \dots, A_j . Como antes, un paréntesis óptimo debe dividir este producto entre A_k y A_{k+1} durante aproximadamente $1 \leq k < j$. A menos que podamos garantizar que k siempre sea igual a $j - 1$, encontraríamos que teníamos sub problemas de la forma A_1, A_2, \dots, A_k y $A_{k+1}, A_{k+2}, \dots, A_j$, y que el último sub problema no es de la forma A_1, A_2, \dots, A_j . Para este problema, necesitábamos permitir que nuestros sub problemas varíen en "ambos extremos", es decir, permitir que tanto i como j varíen en el sub problema A_i, A_{i+1}, \dots, A_j .

La sub estructura óptima varía de un dominio a otro de dos maneras:

1. cuántos subproblemas utiliza una solución óptima para el problema original, y
2. cuántas opciones tenemos para determinar qué subproblema (s) usar en una solución óptima.

Informalmente, el tiempo de ejecución de un algoritmo de programación dinámica depende del producto de dos factores: el número de subproblemas en general y cuántas opciones consideramos para cada subproblema.

Por lo general, el gráfico de subproblema ofrece una forma alternativa de realizar el mismo análisis. Cada vértice corresponde a un subproblema, y las opciones para un subproblema son las aristas incidentes a ese subproblema. Recuerde que en el corte de varillas, el gráfico de subproblema tenía n vértices y, como máximo, n aristas por vértice, produciendo un $O(n^2)$ tiempo de ejecución. Para la multiplicación de la cadena matricial, si dibujáramos el gráfico del subproblema, tendría $O(n^2)$ vértices y cada vértice tendría un grado como máximo $n - 1$, dando un total de $O(n^3)$ vértices y aristas.



La programación dinámica a menudo utiliza una sub estructura óptima de abajo hacia arriba. Es decir, primero encontramos soluciones óptimas para los sub problemas y, una vez resueltos los sub problemas, encontramos una solución óptima para el problema. Encontrar una solución óptima al problema implica elegir entre sub problemas en cuanto

a los que usaremos para resolver el problema. El costo de la solución del problema suele ser el costo del sub problema más un costo que es directamente atribuible a la elección en sí. En el corte de varillas, por ejemplo, primero resolvimos los sub problemas de determinar formas óptimas de cortar varillas de longitud i por $i = 0, 1, \dots, n - 1$, y luego determinamos qué sub problema produjo un óptimo solución para una varilla de longitud n , usando la ecuación de r_n . El costo atribuible a la elección en sí es el término p_i en la ecuación r_n . En la multiplicación de la cadena de la matriz, determinamos paréntesis óptimos de las sub cadenas de A_i, A_{i+1}, \dots, A_j , y luego elegimos la matriz A_k en la cual dividir el producto. El costo atribuible a la elección en sí es el término p_{i-1}, p_k, p_j .

Sub problemas superpuestos

El segundo ingrediente que debe tener un problema de optimización para que se aplique la programación dinámica es que el espacio de los subproblemas debe ser "pequeño" en el sentido de que un algoritmo recursivo para el problema resuelve los mismos subproblemas una y otra vez, en lugar de generar siempre nuevos subproblemas. Típicamente, el número total de subproblemas distintos es un polinomio en el tamaño de entrada. Cuando un algoritmo recursivo revisita el mismo problema repetidamente, decimos que el problema de optimización tiene subproblemas superpuestos. Por el contrario, un problema para el cual es adecuado un enfoque de divide y vencerás generalmente genera problemas nuevos en cada paso de la recursión. Los algoritmos de programación dinámica generalmente aprovechan la superposición de subproblemas resolviendo cada subproblema una vez y luego almacenando la solución en una tabla donde se puede buscar cuando sea necesario, utilizando un tiempo constante por búsqueda.

Reconstruyendo una solución optima.

Como cuestión práctica, a menudo almacenamos la elección que hicimos en cada sub problema en una tabla para que no tengamos que reconstruir esta información a partir de los costos que almacenamos.

Memorización.

Como vimos para el problema del corte de varillas, existe un enfoque alternativo a la programación dinámica que a menudo ofrece la eficiencia del enfoque de programación dinámica ascendente mientras se mantiene una estrategia descendente. La idea es memorizar el algoritmo natural, pero ineficiente, recursivo. Como en el enfoque de abajo hacia arriba, mantenemos una tabla con soluciones de sub problemas, pero la estructura de control para completar la tabla es más parecida al algoritmo recursivo.

Un algoritmo recursivo memorizado mantiene una entrada en una tabla para la solución de cada sub problema. Cada entrada de la tabla contiene inicialmente un valor especial para indicar que la entrada aún no se ha completado. Cuando el sub problema se encuentra por primera vez a medida que se desarrolla el algoritmo recursivo, su solución se calcula y luego se almacena en la tabla. Cada vez que nos encontramos con este sub problema, simplemente buscamos el valor almacenado en la tabla y lo devolvemos.

En la práctica general, si todos los sub problemas deben resolverse al menos una vez, un algoritmo de programación dinámica ascendente generalmente supera al algoritmo memorizado descendente correspondiente en un factor constante, porque el algoritmo ascendente no tiene sobrecarga para la recursión y menos sobrecarga para mantener la tabla. Además, para algunos problemas podemos explotar el patrón regular de accesos a la tabla en el algoritmo de programación dinámica para reducir aún más los requisitos de tiempo o espacio. Alternativamente, si algunos sub problemas en el espacio de sub problemas no necesitan ser resueltos en absoluto, la solución memorizada tiene la ventaja de resolver solo aquellos sub problemas que definitivamente se requieren.

Para construir un algoritmo de programación dinámica, se pueden seguir los siguientes pasos:

- Caracterizar la estructura de una solución optima.
- Recursivamente, definir el valor de una solución optima.

- Calcular el valor de una solución optima, típicamente de abajo hacia arriba (es decir, comenzando por los casos base y siguiendo la estructura de la solución optima hasta alcanzar el valor buscado).
- Construir una solución optima mediante la información calculada.

Los pasos 1–3 forman la base de una solución de programación dinámica para un problema. Si solo necesitamos el valor de una solución óptima, y no la solución en sí, entonces podemos omitir el paso 4. Cuando realizamos el paso 4, a veces mantenemos información adicional durante el paso 3 para que podamos construir fácilmente una solución óptima (por ejemplo, cuando buscamos la cantidad minima de pasos desde un punto a otro, teniendo varios caminos posibles. Podemos quedarnos con el minimo valor de pasos o pretender que se demuestre cual es el camino que une los dos puntos en esa cantidad de pasos).

Otro ingrediente recurrente en los problemas de resolución dinámica es la **ecuación de recurrencia**. Se obtiene en parte por la naturaleza recursiva de los algoritmos de este tipo. Nos puede servir tanto como para definir los casos base y los recursivos de la solución, como la determinación de la complejidad espacial y temporal de la solución.

A continuacion, analizaremos una serie de problemas resueltos mediante programacion dinamica, intentando comprender de forma mas practica, como generar algoritmos mediante esta tecnica.

El problema del corte de varillas.

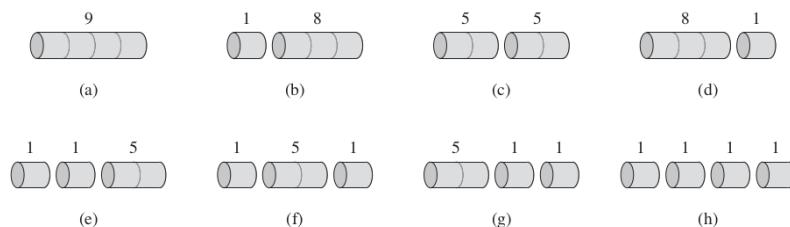
Nuestro primer ejemplo usa programación dinámica para resolver un problema simple al decidir dónde cortar las varillas de acero. Serling Enterprises compra barras de acero largas y las corta en barras más cortas, que luego vende. Cada corte es gratis. La gerencia de Serling Enterprises quiere saber la mejor manera de cortar las barras.

Suponemos que sabemos, para $i = 1, 2, \dots$ el precio p_i en dólares que Serling Enterprises cobra por una barra de longitud i pulgadas. Las longitudes de varilla son siempre un número entero de pulgadas.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

El problema del corte de varilla es el siguiente. Dada una barra de longitud n pulgadas y una tabla de precios p_i para $i = 1, 2, \dots, n$, determine el ingreso máximo que puede obtener cortando la barra y vendiendo las piezas. Tenga en cuenta que si el precio p_n para una barra de longitud n es lo suficientemente grande, una solución óptima puede no requerir corte alguno.

Considere el caso cuando $n = 4$. La figura siguiente muestra todas las formas de cortar una varilla de 4 pulgadas de largo, incluida la forma sin cortes. Vemos que cortar una varilla de 4 pulgadas en dos piezas de 2 pulgadas produce ingresos $p_2 + p_2 = 5 + 5 = 10$, lo cual es óptimo.



Podemos cortar una varilla de longitud n de 2^{n-1} de diferentes maneras, ya que tenemos la opción independiente de cortar, o no cortar, a una distancia de 1 pulgada del extremo izquierdo.

Para $i = 1, 2, \dots, n - 1$, denotamos una descomposición en piezas utilizando la notación aditiva ordinaria, de modo que $7 = 2 + 2 + 3$ indica que una barra de longitud 7 se corta en tres piezas: dos de longitud 2 y una de longitud 3. Si una solución óptima corta la barra en k piezas, durante $1 \leq k \leq n$, luego una descomposición óptima:

$$n = i_1 + i_2 + \dots + i_k$$

de la barra en trozos de longitudes i_1, i_2, \dots, i_k proporciona los ingresos máximos correspondientes

$$r = p_1 + p_2 + \dots + p_k$$

Para nuestro problema de muestra, podemos determinar las cifras de ingresos óptimas r_i , para $i = 1, 2, \dots, 10$, mediante inspección, con las correspondientes descomposiciones óptimas

- r 1 = 1 de la solución 1 = 1 (sin cortes),
- r 2 = 5 de la solución 2 = 2 (sin cortes),
- r 3 = 8 de la solución 3 = 3 (sin cortes),
- r 4 = 10 de la solución 4 = 2 + 2,
- r 5 = 13 de la solución 5 = 2 + 3,
- r 6 = 17 de la solución 6 = 6 (sin cortes),
- r 7 = 18 de la solución 7 = 1 + 6 o 7 = 2 + 2 + 3,
- r 8 = 22 de la solución 8 = 2 + 6,
- r 9 = 25 de la solución 9 = 3 + 6,
- r 10 = 30 de la solución 10 = 10 (sin cortes),

En términos más generales, podemos enmarcar los valores r_n para $n \geq 1$ en términos de ingresos óptimos de barras más cortas:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

El primer argumento, p_n , corresponde a no hacer ningún corte y vender la barra de longitud n tal como está. Los otros argumentos $n - 1$ para max corresponden al ingreso máximo obtenido al hacer un corte inicial de la barra en dos piezas de tamaño i y $n - i$, para cada $i = 1, 2, \dots, n - 1$ y luego cortar de manera óptima esas piezas aún más, obteniendo ingresos r_i y r_{n-i} de esas dos piezas. Como no sabemos de antemano qué valor de i optimiza los ingresos, tenemos que considerar todos los valores posibles para i y elegir el que maximice los ingresos. También tenemos la opción de elegir no i si podemos obtener más ingresos vendiendo la barra sin cortar. Tenga en cuenta que para resolver el problema original de tamaño n , resolvemos problemas más pequeños del mismo tipo, pero de tamaños más pequeños. Una vez que hacemos el primer corte, podemos considerar las dos piezas como instancias independientes del problema de corte de varillas.

La solución óptima general incorpora soluciones óptimas a los dos subproblemas relacionados, maximizando los ingresos de cada una de esas dos piezas. Decimos que el problema del corte de varillas exhibe una subestructura óptima: las soluciones óptimas a un problema incorporan soluciones óptimas a subproblemas relacionados, que podemos resolver de forma independiente.

En un relacionado, pero un poco más simple, una forma de organizar una estructura recursiva para el problema del corte de varilla, vemos una descomposición que consiste en una primera pieza de longitud que corté el extremo izquierdo, y luego un resto de longitud derecha $n - i$. Solo el resto, y no la primera pieza, puede dividirse aún más. Podemos ver cada descomposición de una varilla de longitud n de esta manera: como una primera pieza seguida de cierta descomposición del resto. Al hacerlo, podemos poner la solución sin cortes en absoluto, diciendo que la primera pieza tiene un tamaño $i = n$ y ingresos p_n . Podemos ver cada descomposición de una varilla de longitud n de esta manera: como una primera pieza seguida de cierta descomposición del resto. Al hacerlo, podemos poner la solución sin cortes en absoluto, diciendo que la primera pieza tiene un tamaño $i = n$ y ingresos p_n y que el resto tiene un tamaño 0 con los ingresos correspondientes $r_0 = 0$. De este modo obtenemos la siguiente versión más simple de la ecuación:

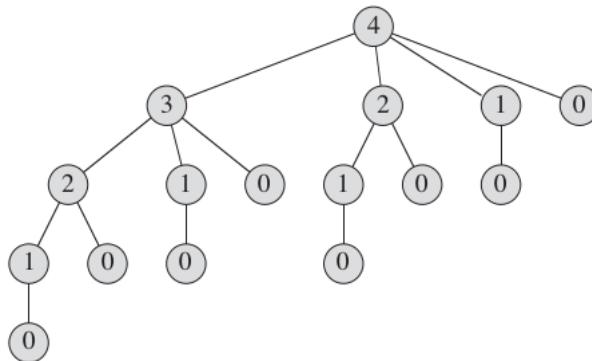
$$r_n = \max(p_i, r_{n-i}) \forall 1 \leq i \leq n$$

En esta formulación, una solución óptima incorpora la solución a un solo subproblema relacionado, el resto, en lugar de dos.

Solución recursiva top-down

```
Sea la función recursiva CUT_ROD(precios, longitud):
    Si longitud == 0
        Devolver 0.
    Sea q la variable inicializada con -INFINITO.
    Para todos los valores i entre 0 y longitud-1:
        q = max(q, precios[i] + ROC_CUT(precios, longitud-i)).
    Devolver q.
```

¿Por qué CUT-ROD es tan ineficiente? El problema es que CUT-ROD se llama a sí mismo recursivamente una y otra vez con los mismos valores de parámetros; resuelve los mismos subproblemas repetidamente.



Ahora discutiremos cómo convertir CUT-ROD en un algoritmo eficiente, usando programación dinámica.

El método de programación dinámica funciona de la siguiente manera. Habiendo observado que una solución recursiva ingenua es ineficiente porque resuelve los mismos subproblemas repetidamente, organizamos para que cada subproblema se resuelva solo una vez, guardando su solución. Si necesitamos referirnos a la solución de este subproblema nuevamente más tarde, simplemente podemos buscarlo, en lugar de volver a calcularlo. La programación dinámica utiliza memoria adicional para ahorrar tiempo de cálculo; Sirve un ejemplo de una compensación de memoria de tiempo . El ahorro puede ser dramático: una solución de tiempo exponencial se puede transformar en una solución de tiempo polinomial. Un enfoque de programación dinámica se ejecuta en tiempo polinómico cuando el número de subproblemas distintos involucrados es polinomial en el tamaño de entrada y podemos resolver cada uno de estos subproblemas en tiempo polinómico.

Por lo general, hay dos formas equivalentes de implementar un enfoque de programación dinámica. Los ilustraremos con nuestro ejemplo de corte de varillas.

El primer enfoque es de arriba hacia abajo con la memorización . En este enfoque, escribimos el procedimiento de forma recursiva de forma natural, pero modificado para guardar el resultado de cada subproblema (generalmente en una matriz o tabla hash). El procedimiento ahora primero verifica si ha resuelto previamente este subproblema. Si es así, devuelve el valor guardado, guardando más cálculos en este nivel; si no, el procedimiento calcula el valor de la manera habitual. Decimos que el procedimiento recursivo ha sido memorizado ; "recuerda" qué resultados ha calculado previamente.

El segundo enfoque es el método ascendente . Este enfoque generalmente depende de alguna noción natural del "tamaño" de un subproblema, de modo que la resolución de cualquier subproblema en particular depende solo de la resolución de subproblemas "más pequeños". Clasificamos los subproblemas por tamaño y los resolvemos en orden de tamaño, los más pequeños primero. Al resolver un subproblema en particular, ya hemos

resuelto todos los subproblemas más pequeños de los que depende su solución, y hemos guardado sus soluciones. Resolvemos cada subproblema solo una vez, y cuando lo vemos por primera vez, ya hemos resuelto todos sus subproblemas de requisitos previos.

Estos dos enfoques producen algoritmos con el mismo tiempo de ejecución asintótico , excepto en circunstancias inusuales donde el enfoque de arriba hacia abajo no se repite para examinar todos los posibles subproblemas. El enfoque ascendente a menudo tiene factores constantes mucho mejores, ya que tiene menos sobrecarga para las llamadas a procedimientos. Aquí está el pseudocódigo para el procedimiento CUT-ROD de arriba hacia abajo, con la memoria agregada:

```
BOTTOM-UP-CUT-ROD (p, n)
    sea r [n] una nueva matriz
    r [0] = 0
    para j = 1 a n
        q = -infinito
        para i = 1 a j
            q = max (q, p [i] + r [j-i])
        r [j] = q
    retorno r [n]
```

El tiempo de ejecución de este algoritmo es de tan solo n^2 . Para reconstruir la solución, se puede hacer uso del siguiente algoritmo.

```
varilla de corte de abajo hacia arriba extendida (p, n, r, s)
    dejemos que r & s sea una nueva matriz de tamaño 0, ..., n
    r [0] = 0 y s [0] = 0
    para j = 1 a n:
        q = -infinito
        para i = 1 a j do
            si q < p [i] + r [j-i]
                q = p [i] + r [j-i]
                s [j] = i
        r [j] = n

    Devolver r, s
```

Nota: Pensar el problema primero por fuerza bruta, puede servir para dar cuenta de las decisiones que se deben tomar. Despues de todo, la solucion por fuerza bruta siempre da el optimo, el problema es que debe ser mejorado, en este caso, mediante la memorización.

Weigthed Interval Scheduling

Hemos visto que un algoritmo codicioso en particular produce una solución óptima al problema de programación de intervalos, donde el objetivo es aceptar un conjunto tan grande de intervalos que no se superpongan como sea posible. El problema de programación de intervalos ponderados es una versión estrictamente más general, en la que cada intervalo tiene un valor (o peso) determinado, y queremos aceptar un conjunto de valor máximo.

Diseño del algoritmo recursivo

Dado que el problema de programación de intervalos original es simplemente el caso especial en el que todos los valores son iguales a 1, sabemos que la mayoría de los algoritmos codiciosos no resolverán este problema de manera óptima. Incluso el algoritmo que funcionó antes (eligiendo repetidamente el intervalo que termina antes) ya no es óptimo en esta configuración más general.

Usamos la notación de nuestra discusión sobre la programación de intervalos. Tenemos n solicitudes etiquetadas como $1, \dots, n$, con cada solicitud i especificando una hora de inicio s_i y una hora de finalización f_i . Cada intervalo i ahora también tiene un valor, o peso v_i . Dos intervalos son compatibles si no se superponen. El objetivo de nuestro

problema actual es seleccionar un subconjunto S de intervalos mutuamente compatibles, para maximizar la suma de los valores de los intervalos seleccionados.

Supongamos que las solicitudes se ordenan por tiempo de finalización no decreciente: $f_1 \leq f_2 \leq \dots \leq f_n$. Diremos que una solicitud i viene antes que una solicitud j si $i < j$. Este será el orden natural de izquierda a derecha en el que consideraremos los intervalos. Para ayudar a hablar de este orden, definimos $p(j)$, para un intervalo j , como el índice más grande $i < j$ tal que los intervalos i y j son disjuntos. En otras palabras, i es el intervalo más a la izquierda que termina antes de que comience j . Definimos $p(j) = 0$ si ninguna solicitud $i < j$ es disjunta de j . En la Figura se muestra un ejemplo de la definición de $p(j)$.

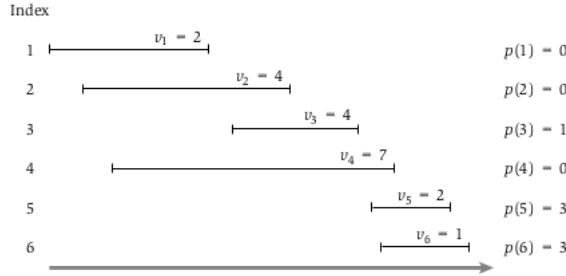


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j .

Ahora, dada una instancia del problema de programación de intervalos ponderados, consideraremos una solución óptima O , ignorando por ahora que no tenemos idea de qué es. Aquí hay algo completamente obvio que podemos decir acerca de O : o el intervalo n (el último) pertenece a O , o no. Supongamos que exploramos un poco más ambos lados de esta dicotomía. Si $n \in O$, entonces claramente ningún intervalo indexado estrictamente entre $p(n)$ y n puede pertenecer a O , porque por la definición de $p(n)$, sabemos que los intervalos $p(n) + 1, p(n) + 2, \dots, n - 1$ se superponen con n . Además, si $n \in O$, entonces O debe incluir una solución óptima al problema que consta de solicitudes $1, \dots, p(n)$, porque si no fuera así, podríamos reemplazar la elección de O de solicitudes de $1, \dots, p(n)$ con uno mejor, sin peligro de superposición de la solicitud n .

Por otro lado, si $n \notin O$, entonces O es simplemente igual a la solución óptima al problema que consiste en solicitudes $1, \dots, n - 1$. Esto es mediante un razonamiento completamente análogo: asumimos que O no incluye la solicitud n ; por lo que si no elige el conjunto óptimo de solicitudes de $1, \dots, n - 1$, podríamos reemplazarlo por uno mejor.

Todo esto sugiere que encontrar la solución óptima en los intervalos $1, 2, \dots, n$ implica buscar las soluciones óptimas de problemas más pequeños de la forma $1, 2, \dots, j$. Por lo tanto, para cualquier valor de j entre 1 y n , denote O_j la solución óptima al problema que consta de solicitudes $1, \dots, j$, y sea $OPT(j)$ el valor de esta solución. (Definimos $OPT(0) = 0$, basándonos en la convención de que este es el óptimo sobre un conjunto vacío de intervalos.) La solución óptima que estamos buscando es precisamente O_n , con valor $OPT(n)$. Para la solución óptima O_j en $1, 2, \dots, j$, nuestro razonamiento anterior (generalizando a partir del caso en el que $j = n$) dice que $j \in O_j$, en cuyo caso $OPT(j) = v_j + OPT(p(j))$, o $j \notin O_j$, en cuyo caso $OPT(j) = OPT(j - 1)$. Dado que estas son precisamente las dos opciones posibles ($j \in O_j$ o $j \notin O_j$), podemos decir además que

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$$

```

Compute-Opt( j )
  If j = 0 then
    Return 0
  Else
    Return max(v j + Compute-Opt(p(j)), Compute-Opt (j - 1))
  Endif

```

Sin embargo, la implementación de esta solución es intractable, ya que corre en un tiempo exponencial. Esta es una buena oportunidad para aplicar memorización para reducir su costo.

La memorización de la recursión

De hecho, no estamos tan lejos de tener un algoritmo de tiempo polinomial. Una observación fundamental, que forma el segundo componente crucial de una solución de programación dinámica, es que nuestro algoritmo recursivo Compute-Opt en realidad solo resuelve $n + 1$ subproblemas diferentes: Compute-Opt (0), Compute-Opt (1), ..., Compute-Opt (n). El hecho de que se ejecute en tiempo exponencial tal como está escrito se debe simplemente a la espectacular redundancia en la cantidad de veces que emite cada una de estas llamadas.

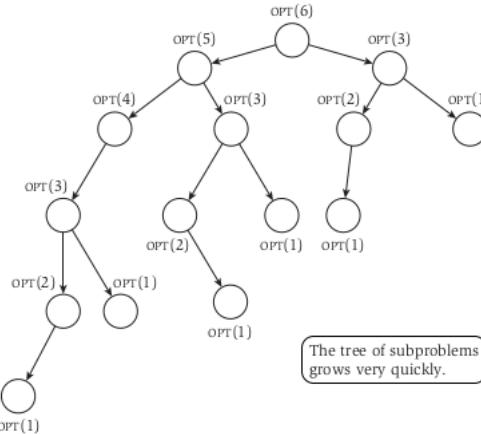


Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

¿Cómo podríamos eliminar toda esta redundancia? Podríamos almacenar el valor de Compute-Opt en un lugar accesible globalmente la primera vez que lo calculemos y luego simplemente usar este valor precalculado en lugar de todas las llamadas recursivas futuras. Esta técnica de guardar valores que ya se han calculado se denomina memoria.

Implementamos la estrategia anterior en el procedimiento más “inteligente” M-Compute-Opt. Este procedimiento utilizará una matriz $M[0 \dots n]$; $M[j]$ comenzará con el valor “vacío”, pero mantendrá el valor de $Compute - Opt(j)$ tan pronto como se determine por primera vez. Para determinar $OPT(n)$, invocamos $M - Compute - Opt(n)$.

```

M-Compute-Opt( j )
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
    Define M[j] = max(v_j + M-Compute-opt (p(j)) , M-Compute-
    opt (j - 1))
    Return M[j]
  Endif
  
```

Publicidad en carretera

Enunciado

Suponga que está gestionando la construcción de vallas publicitarias en la Stephen Daedalus Memorial Highway, un tramo de carretera muy transitado que corre de oeste a este por M millas. Los posibles lugares para las vallas publicitarias están dados por los números x_1, x_2, \dots, x_n , cada uno en el intervalo $[0, M]$ (especificando su posición a lo largo de la carretera, medida en millas desde su extremo occidental). Si coloca una valla publicitaria en la ubicación x_i , recibe un ingreso de $r_i > 0$.

Las regulaciones impuestas por el Departamento de Carreteras del condado requieren que no haya dos vallas publicitarias a menos de o igual a 5 millas entre sí. Le gustaría colocar vallas publicitarias en un subconjunto de los sitios para maximizar sus ingresos totales, sujeto a esta restricción.

Ejemplo: Suponga que $M = 20$, $n = 4$, $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$ y $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$

Entonces, la solución optima para este caso seria colocar las vallas en los puntos x_1 y x_3 por un ingreso total de 10.

Genere un algoritmo que tome una instancia del problema y devuelva el maximo de ingreso que puede ser obtenido de cualquier subconjunto de los lugares posibles. El algoritmo debe ser eficiente.

Diseño de la solución

Naturalmente, podemos aplicar la programación dinámica a este problema si razonamos de la siguiente manera. Considere una solución óptima para una instancia de entrada determinada; en esta solución, colocamos una valla publicitaria en el sitio x_n o no. Si no lo hacemos, la solución óptima en sitios x_1, \dots, x_n es realmente la misma que la solución óptima en los sitios x_1, \dots, x_{n-1} ; si lo hacemos, entonces deberíamos eliminar x_n y todos los demás sitios que se encuentran dentro de un radio de 5 millas y encontrar una solución óptima en lo que queda. El mismo razonamiento se aplica cuando miramos el problema definido solo por los primeros j sitios, x_1, \dots, x_j o incluimos x_j en la solución óptima o no lo hacemos, con las mismas consecuencias. Definamos alguna notación para ayudar a expresar esto. Para un sitio x_j , dejamos que $e(j)$ denote el sitio más al este de x_i que está a más de 5 millas de x_j . Dado que los sitios están numerados de oeste a este, esto significa que los sitios $x_1, x_2, \dots, x_{e(j)}$ siguen siendo opciones válidas una vez que hemos elegido colocar una valla publicitaria en x_j , pero los sitios $x_{e(j)+1}, \dots, x_{j-1}$ no lo son. Ahora, nuestro razonamiento anterior justifica la siguiente repetición. Si dejamos que $OPT(j)$ denote los ingresos del subconjunto óptimo de sitios entre x_1, \dots, x_j , entonces tenemos

$$OPT(j) = \max(r_j + OPT(e(j)), OPT(j - 1))$$

Ahora tenemos la mayoría de los ingredientes que necesitamos para un algoritmo de programación dinámica. Primero, tenemos un conjunto de n subproblemas, que consta de los primeros j sitios para $j = 0, 1, 2, \dots, n$. En segundo lugar, tenemos una recurrencia que nos permite construir las soluciones a los subproblemas, dada por $OPT(j) = \max(r_j + OPT(e(j)), OPT(j - 1))$. Para convertir esto en un algoritmo, solo necesitamos definir una matriz M que almacenará los valores OPT y lanzará un bucle alrededor de la recurrencia que construya los valores $M[j]$ en orden creciente de j .

```
Inicializamos M[0] = 0 y M[1] = r1
Desde j = 2, 3, ..., n:
    Calcular M[j] usando la recurrencia
    Devolver M[n]
```

Al igual que con todos los algoritmos de programación dinámica que hemos visto en este capítulo, se puede encontrar un conjunto óptimo de vallas publicitarias rastreando los valores en la matriz M . Dados los valores $e(j)$ para todo j , el tiempo de ejecución del algoritmo es $O(n)$, ya que cada iteración del ciclo toma un tiempo constante. También podemos calcular todos los valores de $e(j)$ en el tiempo $O(n)$ de la siguiente manera. Para cada ubicación de sitio x_i , definimos $x_i = x_i - 5$. Luego fusionamos la lista ordenada x_1, \dots, x_n con la lista ordenada x_1, \dots, x_n en tiempo lineal. Ahora examinamos esta lista combinada; cuando llegamos a la entrada x_j , sabemos que cualquier cosa desde este punto en adelante hasta x_j no se puede elegir junto con x_j (ya que está dentro de las 5 millas), por lo que simplemente definimos $e(j)$ como el valor más grande de i para que hemos visto x_i en nuestro escaneo.

El Problema del Viajante de Comercio o TSP

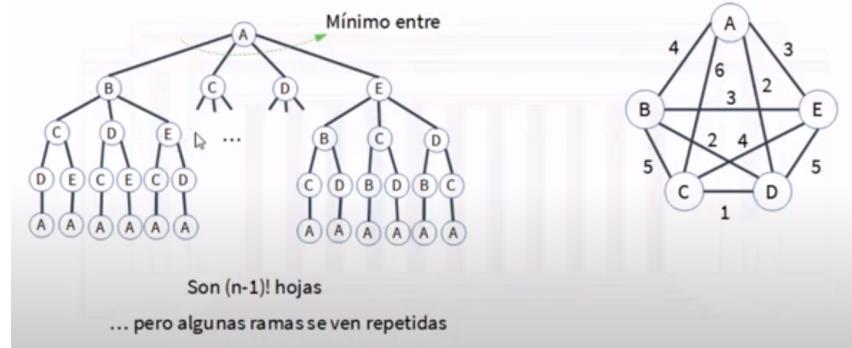
Enunciado

El problema del viajante se puede representar con un grafo $G = (V, E)$ completo con pesos en cada una de sus aristas. Queremos calcular un ciclo o camino P de costo mínimo que inicia y finaliza en una misma ciudad, y pasa solo una vez por cada uno de los nodos.

Diseño de la solución

Si pensamos en fuerza bruta, deberíamos analizar todas las posibles permutaciones de ciudades y calcular para cada una de ellas el costo total. Esto tiene un orden $O(n!)$.

Sin embargo, podemos descomponer el problema tal como lo muestra la figura. Si hacemos esto, notaremos que ciertos problemas se repiten, lo que podríamos usar a nuestro favor aplicando la memorización de la programación dinámica.



Por dar un ejemplo, sea un conjunto de cinco ciudades $\{a, b, c, d, e\}$, nuestro ciclo empieza y termina en la ciudad a, entonces podemos decir que costo mínimo del ciclo es igual a

$OPT(a, \{b, c, d, e\}) = \min(w(b, a) + OPT(b, \{c, d, e\})), w(c, a) + OPT(c, \{b, d, e\}), w(d, a) + OPT(d, \{b, c, e\})$

que definitiva se ve podria resumir en la siguiente ecuación de recurrencia.

$$OPT(i, \{S\}) = \min_{j \in S} w(i, j) + OPT(j, \{S - j\})$$

Con S como el conjunto de nodos e i la ciudad donde se esta. A continuacion mostramos la solución en un algoritmo iterativo.

```
// ciudad 1 es la ciudad inicial
Por cada ciudad i = 2 hasta n:
    OPT[i][0] = w[i][1]

Desde k = 1 a n-2:
    Para todo subset S de C-{1} de tamaño k.
        Para cada elemento i de S:
            OPT[i, S - i] = INFINITO
            Por cada elemento j de S - i:
                r = OPT[j, S - {i, j}] + w[j][i]

                Si r < OPT[i, S - i]:
                    OPT[i, S - i] = r

    P = INFINITO //costo del camino minimo
    Desde j=2 a n:
        ciclo = OPT[j, S - {1, j}] + w[1][i]
        Si P > ciclo:
            P = ciclo

    retornar P
```

Esta solución comprende un orden de ejecución igual a $O(n^2 2^n)$ que no es polinomial pero es mucho mejor que la solución por fuerza bruta.

Camibio mínimo en monedas

Enunciado

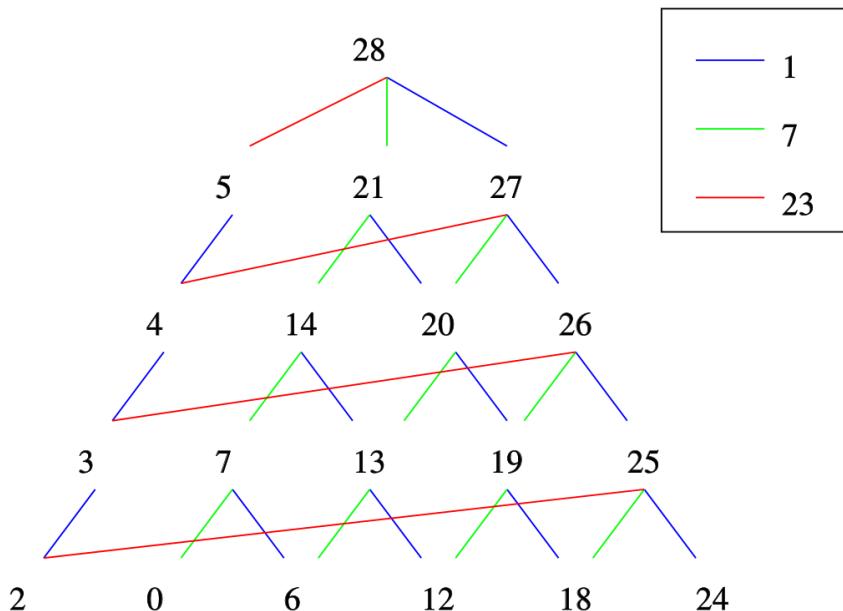
Contamos con un conjunto de monedas $S = \{c_1, c_2, \dots, c_n\}$ de diferente denominación sin restricción de cantidad. Dado un importe X de cambio a dar, queremos entregar la menor cantidad de monedas posible como cambio.

Diseño de la solución

Una solución por fuerza bruta comprendería evaluar para el valor X , las posibles combinaciones de cantidad de monedas de cada denominación. Un enfoque más visual podría ser el siguiente: para cada denominación c_i , calcular el mínimo de monedas a entregar si el valor fuese $X - c_i$ y a ese valor sumar uno. Esto se puede expresar en una ecuación de recurrencia

$$OPT(0) = 0 // OPT(X) = \min_{c \in S} (OPT(X - c)) + 1$$

Claro que esto tiene un costo $O(X^n)$, pero notamos que el cálculo de algunos optimos se repiten, lo cual es un tanto obvio. Prestemos un poco de atención al ejemplo que tenemos en la figura. Son tres denominaciones $S = \{1, 7, 23\}$ y un valor $X = 28$.



A continuación elaboramos un algoritmo iterativo cuya memorización es de abajo hacia arriba.

```

opt[0] = 0
elegido[0] = 0

Con i desde 1 hasta n:
    minimo = INF
    elegido[i] = 0

    Con j desde 1 hasta n:
        resto = x - c[j]
        si resto >= 0 y minimo > opt[resto]
            elegido[i] = j
            minimo = opt[resto]

    opt[i] = i + minimo

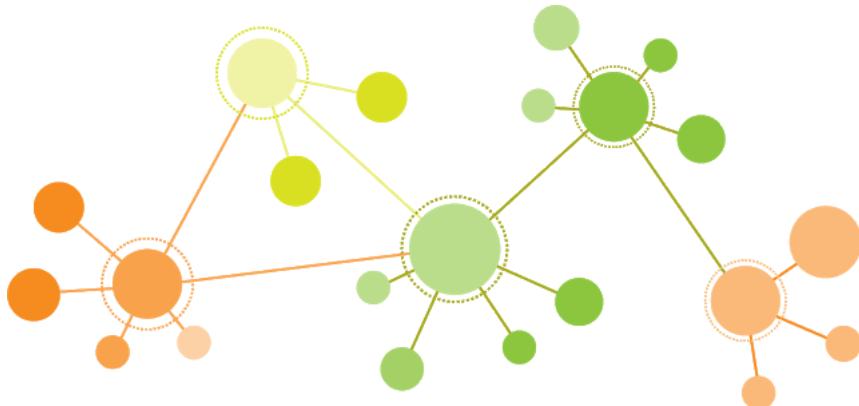
    resto = x
    mientras resto > 0:
        imprimir c[elegido[resto]]
        resto = resto - c[elegido[resto]]

imprimir opt[x]

```

El tiempo de ejecución de este algoritmo es $O(n^x)$ mientras que ocupa un espacio igual a $O(x)$. Esto lo transforma en un algoritmo pseudopolinómico.

Flujo de redes.



Así como podemos modelar un mapa de ruta como un gráfico dirigido para encontrar el camino más corto de un punto a otro, también podemos interpretar un gráfico dirigido como una "red de flujo" y usarlo para responder preguntas sobre flujos de materiales. Imagine un material que fluye a través de un sistema desde una fuente, donde se produce el material, hasta un sumidero, donde se consume. La fuente produce el material a una velocidad constante, y el sumidero consume el material a la misma velocidad. El "flujo" del material en cualquier punto del sistema es intuitivamente la velocidad a la que se mueve el material. Las redes de flujo pueden modelar muchos problemas, incluidos líquidos que fluyen a través de tuberías, piezas a través de líneas de montaje, corriente a través de redes eléctricas e información a través de redes de comunicación.

Podemos pensar en cada arista dirigido en una red de flujo como un conducto para el material. Cada conducto tiene una capacidad establecida, dada como una velocidad máxima a la que el material puede fluir a través del conducto, como 200 galones de líquido por hora a través de una tubería o 20 amperios de corriente eléctrica a través de un cable. Los vértices son uniones de conducto, y además de la fuente y el sumidero, el material fluye a través de los vértices sin acumularse en ellos. En otras palabras, la velocidad a la que el material entra en un vértice debe ser igual a la velocidad a la que sale del vértice. Llamamos a esta propiedad "conservación del flujo", y es equivalente a la ley actual de Kirchhoff cuando el material es corriente eléctrica.

En el problema de flujo máximo, deseamos calcular la mayor velocidad a la que podemos enviar material desde la fuente al sumidero sin violar ninguna restricción de capacidad. Es uno de los problemas más simples relacionados con las redes de flujo y, como veremos en este capítulo, este problema puede resolverse mediante algoritmos eficientes. Además, podemos adaptar las técnicas básicas utilizadas en los algoritmos de flujo máximo para resolver otros problemas de flujo de red.

Este capítulo presenta dos métodos generales para resolver el problema de flujo máximo. La primera sección se formaliza las nociones de redes y flujos de flujo, definiendo formalmente el problema del flujo máximo. La segunda sección se describe el método clásico de Ford y Fulkerson para encontrar los flujos máximos.

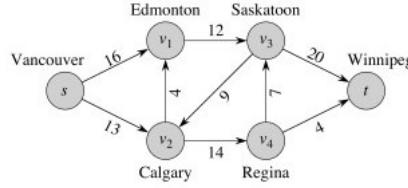
Marco teórico para el Flujo de Redes.

En esta sección, damos una definición teórica de gráficos de redes de flujo, discutimos sus propiedades y definimos el problema de flujo máximo con precisión. También presentamos alguna notación útil.

Redes de flujo y flujos

Una red de flujo $G = (V, E)$ es un gráfico dirigido en el que cada arista $(u, v) \in E$ tiene una capacidad no negativa $c(u, v) \geq 0$. Requerimos además que si E contiene un arista (u, v) , entonces no hay arista (v, u) en la dirección inversa. (En breve veremos cómo solucionar esta restricción). Si $(u, v) \notin E$, entonces por conveniencia definimos $c(u, v) = 0$, y no permitimos los self-loops. Distinguimos dos vértices en una red de flujo:

una fuente s y un sumidero t . Por conveniencia, asumimos que cada vértice se encuentra en algún camino desde la fuente hasta el sumidero. Es decir, para cada vértice $v \in V$, la red de flujo contiene una ruta $s \rightarrow v \rightarrow t$. Por lo tanto, el gráfico está conectado y, dado que cada vértice que no sea s tiene al menos un arista de entrada, $|E| \geq |V| - 1$.



Ahora estamos listos para definir flujos más formalmente. Deje $G = (V, E)$ ser una red de flujo con una función de capacidad c . Seamos la fuente de la red y seamos el sumidero. Un flujo en G es una función de valor real $f : V \times V \rightarrow R$ que satisface las siguientes dos propiedades:

- **Restricción de capacidad:** Para todo $u, v \in V$, se requiere que $0 \leq f(u, v) \leq c(u, v)$.
- **Conservación del flujo:** Para todo $u \in V - \{s, t\}$, se requiere que:

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

Cuando $(u, v) \notin E$, entonces no puede haber flujo de u a v , y $f(u, v) = 0$.

Es decir, que la sumatoria de flujos entrantes en un vértice u es igual a la sumatoria de los flujos salientes del mismo vértice.

Llamamos a la cantidad no negativa $f(u, v)$ el flujo del vértice u al vértice v . El valor $|f|$ de un flujo f se define como:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) = v(f)$$

es decir, el flujo total fuera de la fuente menos el flujo hacia la fuente. (Aquí, la notación $| \cdot |$ denota el valor de flujo, no el valor absoluto o la cardinalidad y en otras bibliotecas lo veremos notado como $v(f)$). Típicamente, una red de flujo no tendrá ningún arista que vaya en dirección a la fuente, y el flujo en la fuente, dado por la suma de $\sum_{v \in V} f(v, s)$, será 0. Sin embargo, lo incluimos, porque cuando introduzcamos redes residuales más adelante en este capítulo, el flujo hacia la fuente será significativo. En el problema de flujo máximo, se nos da una red de flujo G con fuente s y sumidero t , y deseamos encontrar un flujo de valor máximo.

Antes de ver un ejemplo de un problema de flujo de red, exploremos brevemente la definición de flujo y las dos propiedades de flujo. La restricción de capacidad simplemente dice que el flujo de un vértice a otro no debe ser negativo y no debe exceder la capacidad dada. La propiedad de conservación de flujo dice que el flujo total hacia un vértice que no sea la fuente o el sumidero debe ser igual al flujo total que sale de ese vértice, informalmente, "flujo en flujo igual a flujo de salida".

Variantes de Flujo de Redes.

Ejemplo

Una red de flujo puede modelar el problema de transporte que se muestra en la Figura anterior. The Lucky Puck Company tiene una fábrica (fuente) en Vancouver que fabrica discos de hockey, y tiene un almacén (sumidero) en Winnipeg que los almacena. Lucky Puck alquila espacio en camiones de otra empresa para enviar los discos desde la fábrica al almacén. Debido a que los camiones viajan sobre rutas específicas (aristas) entre ciudades (vértices) y tienen una capacidad limitada, Lucky Puck puede enviar como máximo $c(u, v)$ cajas por día entre cada par de ciudades u y v en la Figura anterior. Lucky Puck no tiene control sobre estas rutas y capacidades, por lo que la empresa no puede modificar la red de flujo que se muestra en la Figura anterior. Necesitan determinar la mayor cantidad de cajas por día que pueden enviar y luego producir esta cantidad, ya que no tiene sentido producir más discos de los que pueden enviar a su almacén. Lucky Puck

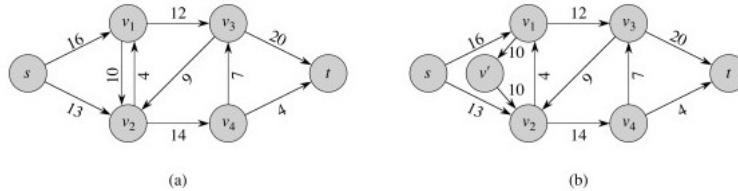
no está preocupado por el tiempo que tarda un disco determinado en llegar de la fábrica al almacén; solo les importa que las cajas que por día salen de la fábrica y lleguen al almacén.

Podemos modelar el "flujo" de envíos con un flujo en esta red porque el número de cajas enviadas por día de una ciudad a otra está sujeto a una restricción de capacidad. Además, el modelo debe obedecer la conservación del flujo, ya que en un estado estable, la velocidad a la que los discos entran en una ciudad intermedia debe ser igual a la velocidad a la que salen. De lo contrario, las cajas se acumularían en las ciudades intermedias.

Problemas de modelado con aristas antiparalelos

Suponga que la empresa de camiones le ofreció a Lucky Puck la oportunidad de alquilar espacio para 10 cajas en camiones que van de Edmonton a Calgary. Parece natural agregar esta oportunidad a nuestro ejemplo y formar la red que se muestra en la Figura siguiente. Sin embargo, esta red tiene un problema: viola nuestra suposición original de que en tanto exista una arista $(v_1, v_2) \in E$, entonces $(v_1, v_2) \notin E$. Llamamos a los dos aristas: (v_1, v_2) y (v_2, v_1) antiparalelas. Por lo tanto, si deseamos modelar un problema de flujo con aristas antiparalelas, debemos transformar la red en una equivalente que no contenga aristas antiparalelas. La Figura siguiente muestra esta red equivalente. Elegimos uno de los dos aristas antiparalelos, en este caso: (v_1, v_2) , y se divide agregando un nuevo vértice v' y reemplazando la arista (v_1, v_2) con el par de aristas $(v_1, v') \text{ y } (v', v_2)$. También establecemos la capacidad de ambos aristas nuevos a la capacidad del arista original. La red resultante satisface la propiedad de que si un arista está en la red, el arista inverso no lo está.

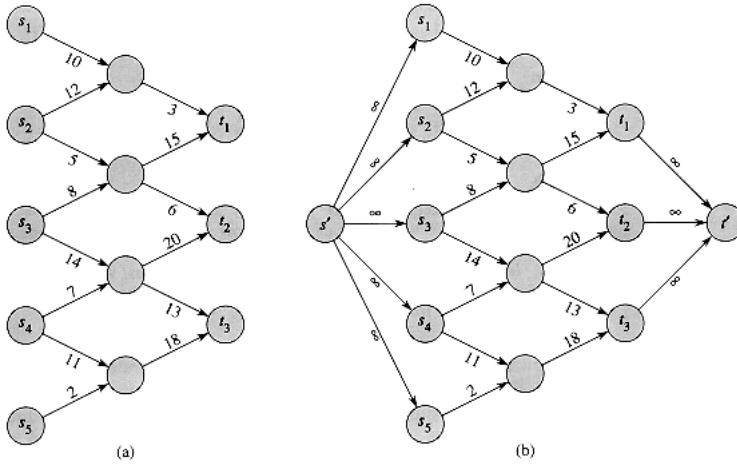
Por lo tanto, vemos que un problema de flujo del mundo real podría ser modelado de forma más natural por una red con aristas antiparalelas. Sin embargo, será conveniente no permitir los aristas antiparalelos, por lo que tenemos una forma sencilla de convertir una red que contiene aristas antiparalelos en una red equivalente sin aristas antiparalelos.



Redes con múltiples fuentes y sumideros.

Un problema de flujo máximo puede tener varias fuentes y sumideros, en lugar de solo uno de cada uno. The Lucky Puck Company, por ejemplo, en realidad podría tener un conjunto de m fábricas $\{s_1, \dots, s_n\}$ y un conjunto de n almacenes $\{t_1, \dots, t_m\}$, como se muestra en la figura siguiente. Afortunadamente, este problema no es más difícil que el flujo máximo ordinario.

Podemos reducir el problema de determinar un flujo máximo en una red con múltiples fuentes y sumideros a un problema de flujo máximo ordinario. La figura siguiente muestra cómo convertir la red de (a) a una red de flujo ordinario con una sola fuente y un solo sumidero. Agregamos una super fuente s y agregamos un arista dirigido (s, s_i) con capacidad $c(s, s_i) = \infty$. También creamos un nuevo super sumidero t y agregamos un arista dirigido (t_i, t) con capacidad $c(t_i, t) = \infty$. Intuitivamente, cualquier flujo en la red en (a) corresponde a un flujo en la red en (b), y viceversa. La fuente única s simplemente proporciona tanto flujo como se desee para las fuentes múltiples s_i , y el sumidero único t también consume tanto flujo como se desee para los sumideros múltiples t_i .



Circulación con demandas

Un caso particular del ejemplo anterior es en el que todos los nodos se ven caracterizados por un cierto factor d_v para todo $v \in V$ que representa la demanda de flujo que posee ese nodo. Si $d_v > 0$ estaremos hablando de un sumidero. Si $d_v < 0$ se tratará de una fuente, finalmente, si $d_v = 0$, se definirá como un nodo intermedio o de conexión. Entonces, tendremos un conjunto de fuentes y otro de sumideros. A estos pueden llegar, tanto aristas entrantes, como de salida.

Dadas estas características, se deben satisfacer las siguientes condiciones.

- Para cada arista $e \in E$, $0 \leq f_e \leq c_e$ (Condición de capacidad)
- Para cada vértice $v \in V$, $d_v = f_{u \in V}(u, v) - f_{u \in V}(v, u)$ (Condición de demanda)

Nuestro problema en este caso es saber si la demanda de cada nodo podrá ser satisfecha. Para que esto sea posible, vamos a requerir que exista igual oferta que demanda. Es decir $\sum_{v \in V} d_v = 0$

Si suponemos que existe una circulación con demanda f factible, entonces se da que:

$$\sum_v d_v = \sum_v (f_{in}(v) - f_{out}(v))$$

Es decir, por cada eje $e(u, v)$ su flujo $f(e)$ se contabiliza dos veces, uno cuando sale de u y otra para cuando entre en v . Por lo tanto la suma total es cero. Entonces podemos afirmar que:

$$\sum_{v:dv>0} d_v = - \sum_{v:dv<0} d_v = D$$

Es decir, por cada eje $e(u, v)$ su flujo $f(e)$ se contabiliza dos veces, uno cuando sale de u y otra para cuando entre en v . Por lo tanto la suma total es cero. Entonces podemos afirmar que:

Para reducir este problema a un problema de redes de flujos, procedemos de la misma forma que en el caso anterior, con la diferencia de que en lugar de que las capacidades de las aristas agregadas se aproximen a infinito, estas serán igual al valor absoluto de la demanda del nodo. Luego procedemos con el algoritmo de Ford-Fulkerson y realizamos un corte dejando el nodo sumidero principal en S y el resto en T .

Una clara consecuencia de esto es que $|f| = D = \sum_s$ fuente $d_s = \sum_t$ sumidero d_t , si esto no se verifica, entonces no hay circulación de flujo posible que cumpla con la demanda requerida.

Circulación con demandas y límites inferiores

Ampliando un poco más el anterior caso, podemos exigir que las aristas, además de tener un límite superior de flujo, dada por la capacidad, ahora tendrán un límite inferior de flujo l_e . Con esto devienen ciertas condiciones a satisfacer.

- Para cada arista $e \in E$, $l_e \leq f_e \leq c_e$ (Condición de capacidad)
- Para cada vértice $v \in V$, $d_v = f_{u \in V}(u, v) - f_{u \in V}(v, u)$ (Condición de demanda)

Vamos a reducir este problema a un problema de circulación con demandas simple. Para esto definimos $l_v = \sum_{u \in V} l(u, v) - \sum_{u \in V} l(v, u)$. Con esto hacemos los siguientes cambios:

- Para cada arista e , se calculará una nueva capacidad $c_e = c_e - l_e$
- Para cada vértice v , se calculará una nueva demanda $d_v = d_v - l_v$
- Eliminamos los límites inferiores

Con estas modificaciones podemos ahora sí, proceder con el algoritmo de Ford-Fulkerson para Circulación con demanda.

El método de Ford-Fulkerson.

Esta sección presenta el método Ford-Fulkerson para resolver el problema de flujo máximo. Lo llamamos un "método" en lugar de un "algoritmo" porque abarca varias implementaciones con diferentes tiempos de ejecución. El método Ford-Fulkerson depende de tres ideas importantes que trascienden el método y son relevantes para muchos algoritmos de flujo y problemas: **redes residuales**, **rutas de aumento** y **cortes**. Estas ideas son esenciales para el importante **teorema de corte mínimo de flujo** **máximo**, que caracteriza el valor de un flujo máximo en términos de cortes de la red de flujo. Terminaremos esta sección presentando una implementación específica del método Ford-Fulkerson y analizando su tiempo de ejecución.

El método Ford-Fulkerson aumenta iterativamente el valor del flujo. Comenzamos con $f(u, v) = 0$ para todos $u, v \in V$, dando un flujo inicial de valor 0. En cada iteración, aumentamos el valor de flujo en G al encontrar una "ruta de aumento" en una "red residual" asociada G_f . Una vez que conocemos las aristas de una ruta de aumento en G_f , podemos identificar fácilmente las aristas específicas en G para los cuales podemos cambiar el flujo de modo que aumentemos el valor del flujo. Aunque cada iteración del método Ford-Fulkerson aumenta el valor del flujo, veremos que el flujo en cualquier arista particular de G puede aumentar o disminuir; Puede ser necesario disminuir el flujo en algunas aristas para permitir que un algoritmo envíe más flujo desde la fuente al sumidero. Aumentamos repetidamente el flujo hasta que la red residual no tenga más rutas de aumento. El teorema de corte mínimo y flujo máximo mostrará que al finalizar, este proceso produce un flujo máximo

```
MetodoDeFordFulkerson(G,s,t):
    Inicializamos el flujo f en 0.
    Mientras que exista una red de aumento p en la red gf:
        Aumentar el flujo f a lo largo de p.
    Devolver f.
```

Para implementar y analizar el método de Ford-Fulkerson, necesitamos introducir algunos conceptos importantes.

Redes residuales.

Intuitivamente, dada una red de flujo G y un flujo f , la red residual G_f consiste en aristas con capacidades que representan cómo podemos cambiar el flujo en las aristas de G . Una arista de la red de flujo puede admitir una cantidad de flujo adicional igual a la capacidad de la arista menos el flujo en esa arista. Si ese valor es positivo, colocamos esa arista en G_f con una "capacidad residual" de $c_f(u, v) = c(u, v) - f(u, v)$. Las únicas aristas de G que están en G_f son aquellos que pueden admitir más flujo; esas aristas (u, v) cuyo flujo es igual a su capacidad tienen $c_f(u, v) = 0$, y no están en G_f .

Sin embargo, la red residual G_f también puede contener aristas que no están en G . Como un algoritmo manipula el flujo, con el objetivo de aumentar el flujo total, es posible que deba disminuir el flujo en una arista en particular. Para representar una posible disminución de un flujo positivo $f(u, v)$ en una arista en G , colocamos una arista (v, u)

en G_f con capacidad residual $c_f(v, u) = f(u, v)$. Es decir, una arista que puede admitir flujo en la dirección opuesta a (u, v) , como máximo cancelando el flujo en (u, v) . Estas aristas inversas en la red residual permiten que un algoritmo envíe el flujo que ya ha enviado a lo largo de una arista. Enviar el flujo de regreso a lo largo de una arista es equivalente a disminuir el flujo en la arista, que es una operación necesaria en muchos algoritmos.

Más formalmente, supongamos que tenemos una red de flujo $G = (V, E)$ con fuente s y sumidero t . Deje f ser un flujo en G , y considere un par de vértices $u, v \in V$. Definimos la capacidad residual $c_f(u, v)$ por

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{si } (u, v) \in E, \\ f(v, u) & \text{si } (v, u) \in E, \\ 0 & \text{en otros casos} \end{cases}$$

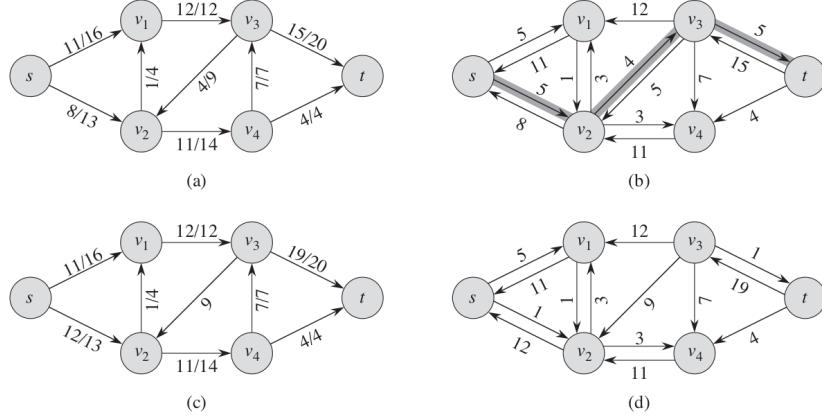


Figure 26.4 (a) The flow network G and flow f of Figure 26.1(b). (b) The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3) , are not shown, a convention we follow in the remainder of this section. (c) The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2) , are labeled only by their capacity, another convention we follow throughout. (d) The residual network induced by the flow in (c).

Debido a nuestra suposición de que $(u, v) \in E$ implica $(v, u) \notin E$, exactamente un caso en la ecuación se aplica a cada par ordenado de vértices.

Como ejemplo de ecuación, si $c(u, v) = 16$ y $f(u, v) = 11$, entonces podemos aumentar $f(u, v)$ por hasta $c_f(u, v) = 5$ unidades antes de que superemos la restricción de capacidad en la arista (u, v) . También deseamos permitir que un algoritmo regrese hasta 11 unidades de flujo de v a u , y por lo tanto $c_f(v, u) = 11$.

Dada una red de flujo $G = (V, E)$ y un flujo f , la **red residual** de G inducida por f es $G_f = (V, E_f)$, donde

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

Es decir, como se prometió anteriormente, cada arista de la red residual, o **arista residual**, puede admitir un flujo que es mayor que 0.

$$|E_f| \leq 2|E|$$

Observe que la red residual G_f es similar a una red de flujo con capacidades dadas por c_f . No satisface nuestra definición de una red de flujo porque puede contener un arista (u, v) y su reversión (v, u) . Aparte de esta diferencia, una red residual tiene las mismas propiedades que una red de flujo, y podemos definir un flujo en la red residual como uno que satisfaga la definición de flujo, pero con respecto a las capacidades c_f en la red G_f .

Un flujo en una red residual proporciona una hoja de ruta para agregar flujo a la red de flujo original. Si f es un flujo en G y f' es un flujo en la red residual correspondiente G_f , definimos $f \uparrow f'$, el **aumento** del flujo f por f' , como una función de $V \times V \rightarrow R$, definida por

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{si } (u, v) \in E, \\ 0 & \text{en otros casos} \end{cases}$$

La intuición detrás de esta definición sigue la definición de la red residual. Aumentamos el flujo en (u, v) por $f'(u, v)$ pero disminuirlo en $f'(v, u)$ porque empujar el flujo en la arista inversa de la red residual significa disminuir el flujo en la red original. El flujo de empuje en la arista inversa de la red residual también se conoce como **cancelación**. Por ejemplo, si enviamos 5 cajas de discos de hockey de u a v y enviamos 2 cajas de v a u , podríamos enviar de manera equivalente (desde la perspectiva del resultado final) 3 cajas de u a v y ninguna de v a u . La cancelación de este tipo es crucial para cualquier algoritmo de flujo máximo.

Lema: Sea $G = (V, E)$ una red de flujo con fuente en s y sumidero en t , y sea f el flujo en G . Sea G_f la red residual de G inducida por f , y sea f' la red de flujo en G_f . Entonces la función $|f \uparrow f'| = |f| + |f'|$ define un flujo en G .

Nota: Se pide encarecidamente no confundir c_f con f' . f' es un flujo que se puede inducir en la red G_f que al igual que en G tiene como límites las capacidades residuales de cada arista. Luego, la inducción del flujo f' tendrá implicancias sobre la red original, que ya definimos como aumento del flujo.

Aumentando caminos.

Dada una red de flujo $G = (V, E)$ y un flujo f , una **ruta de aumento** p es una ruta simple de s a t en la red residual G_f . Según la definición de la red residual, podemos aumentar el flujo en una arista (u, v) de un camino de aumento hasta $c_f(u, v)$ sin violar la restricción de capacidad en cualquiera de (u, v) y (v, u) en la red de flujo original G .

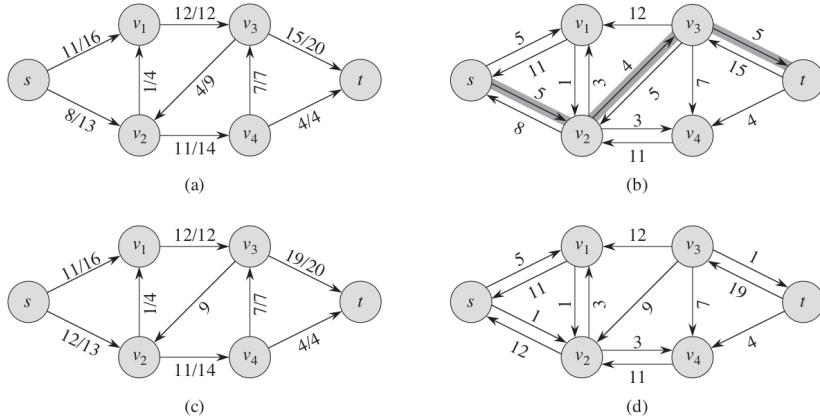


Figure 26.4 (a) The flow network G and flow f of Figure 26.1(b). (b) The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3) , are not shown, a convention we follow in the remainder of this section. (c) The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2) , are labeled only by their capacity, another convention we follow throughout. (d) The residual network induced by the flow in (c).

La ruta sombreada de la figura anterior es una ruta de aumento. Al tratar la red residual G_f en la figura como una red de flujo, podemos aumentar el flujo a través de cada arista de esta ruta hasta en 4 unidades sin violar una restricción de capacidad, ya que la capacidad residual más pequeña en esta ruta es $c_f(v_2, v_4) = 4$. A la cantidad máxima por la que podemos aumentar el flujo en cada arista en una trayectoria de aumento, llamamos la **capacidad residual** de p , dada por

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ está en } p\}$$

Lema: Sea $G = (V, E)$ una red de flujo, sea f el flujo de G , y sea p una ruta de aumento en G_f . Definimos la función $f_p : V \times V \rightarrow R$ como:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{si } (u, v) \text{ está en } p \\ 0 & \text{en otros casos} \end{cases}$$

Entonces, f_p es un flujo de G_f con valor $|f_p| = c_f(p) > 0$

El siguiente corolario muestra que si aumentamos f en f_p , obtenemos otro flujo en G cuyo valor está más cerca del máximo. La Figura (c) muestra el resultado de aumentar el flujo f de la Figura (a) por el flujo f_p en la Figura (b), y la Figura (d) muestra la red residual resultante.

Corolario: Sea $G = (V, E)$ una red de flujo, sea f el flujo de G , y sea p una ruta de aumento en G_f . Sea f_p la función definida anteriormente. Y suponiendo que nosotros incrementamos f por f_p . Entonces la función $f \uparrow f_p$ es un flujo en G con un valor $f \uparrow f_p = |f| + |f_p| > |f|$

Corte de redes de flujo

El método Ford-Fulkerson aumenta repetidamente el flujo a lo largo de trayectorias de aumento hasta que ha encontrado un flujo máximo. ¿Cómo sabemos que cuando termina el algoritmo, realmente hemos encontrado un flujo máximo? El teorema de flujo máximo y corte mínimo, que probaremos en breve, nos dice que un flujo es máximo si y sólo si su red residual no contiene un camino de aumento. Sin embargo, para probar este teorema, primero debemos explorar la noción de corte de una red de flujo.

Un **corte** (S, T) de la red de flujo $G = (V, E)$ es una partición de V en S y $T = V - S$ tal que $s \in S$ y $t \in T$. (Esta definición es similar a la definición de "corte" que usamos para árboles de expansión mínima, excepto que aquí estamos cortando un grafo dirigido en lugar de un grafo no dirigido, e insistimos en que $s \in S$ y $t \in T$.) Si f es un flujo, entonces el flujo neto $f(S, T)$ a través del corte (S, T) se define como

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

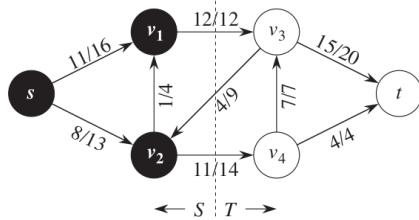


Figure 26.5 A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

La **capacidad** de un corte (S, T)

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Un **corte mínimo** de una red es un corte cuya capacidad es mínima sobre todos los cortes de la red.

La asimetría entre las definiciones de flujo y capacidad de un corte es intencional e importante. Para la capacidad, contamos solo las capacidades de las aristas que van de S a T , ignorando las aristas en la dirección inversa. Para el flujo, consideramos el flujo que va de S a T menos el flujo que va en la dirección inversa de T a S . La razón de esta diferencia se aclarará más adelante en esta sección.

Lema: Sea f un flujo sobre la red de flujos G con fuente en s y sumidero en t , y sea (S, T) un corte cualquiera de G . Entonces el flujo neto que va a través de (S, T) es $f(S, T) = |f|$

Corolario: El valor de cualquier flujo en la red de flujos G está limitada por encima por la capacidad de cualquier corte en G .

El corolario produce la consecuencia inmediata de que el valor de un flujo máximo en una red está limitado desde arriba por la capacidad de un corte mínimo de la red. El importante teorema de corte mínimo de flujo máximo, que ahora declaramos y demostramos, dice que el valor de un flujo máximo es de hecho igual a la capacidad de un corte mínimo.

Teorema del flujo máximo y corte mínimo:

Si f es un flujo en un red de flujo $G = (V, E)$ con fuente en s y sumidero en t , entonces las siguientes condiciones son equivalentes:

- f es el flujo máximo de G .
- La red residual G_f no contiene caminos de aumento.
- $|f| = c(S, T)$ para algún corte (S, T) de G .

El algoritmo basico de Ford-Fulkerson

```
Sea un grafo G = (V, E)
Por cada arista (u, v) en E:
    f(u, v) = 0.

Mientras exista un camino p desde s a t en el grafo residual Gf:
    cf(p) = min{cf(u,v) : (u,v) en p}.
    // Actualizar f por f'.
    Por cada arista (u, v) en p:
        Si (u,v) esta en E:
            f(u,v) = f(u,v) + cf(p).
        Sino:
            f(v,u) = f(v,u) - cf(p).
    Actualizar Gf por Gf'
Retornar f
```

Análisis del algoritmo de Ford-Fulkerson.

Lo primero que debemos revisar es si el algoritmo termina, y para eso, basta con observar dos propiedades de las redes de flujos que usamos de ejemplo. En primera instancia, todos los valores propuestas están en números enteros, eso garantiza que en cada aceleración del algoritmo, el flujo crece. Esto fue aclarado en uno de los corolarios ya mencionados.

Ahora, cuanto mas rápido se incremente el flujo, mas rápido alcanzara su máximo y en la misma medida, el algoritmo finalizara. Pero en el peor de los casos, las capacidades de todas las aristas puede ser de 1, ralentizando el incremento. Sea $C = \sum C_e$ para todos los e salientes de la fuente, vemos que $|f| \leq C$. En cada iteración $|f|$ crece en el peor de los casos, en 1. Por tanto el algoritmo terminara en C iteraciones. Siendo que en cada iteración tiene que buscar caminos por BFS o DFS lo que toma $O(V + E)$ que los podemos aproximar a $O(E)$, actualizar la red de flujos toma $O(V)$ y construir el grafo residual es $O(E)$, se puede estimar que el limite superior para el orden de ejecución del algoritmo es $O(CE)$

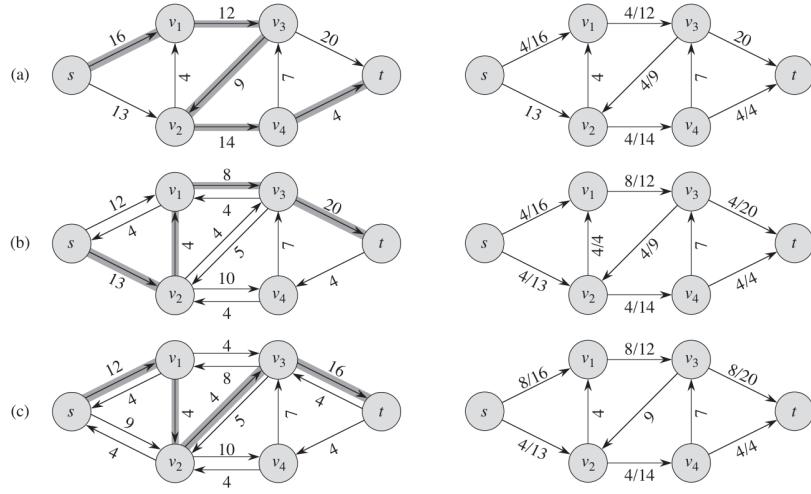


Figure 26.6 The execution of the basic Ford-Fulkerson algorithm. (a)–(e) Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a shaded augmenting path p . The right side of each part shows the new flow f that results from augmenting f by f_p . The residual network in (a) is the input network G .

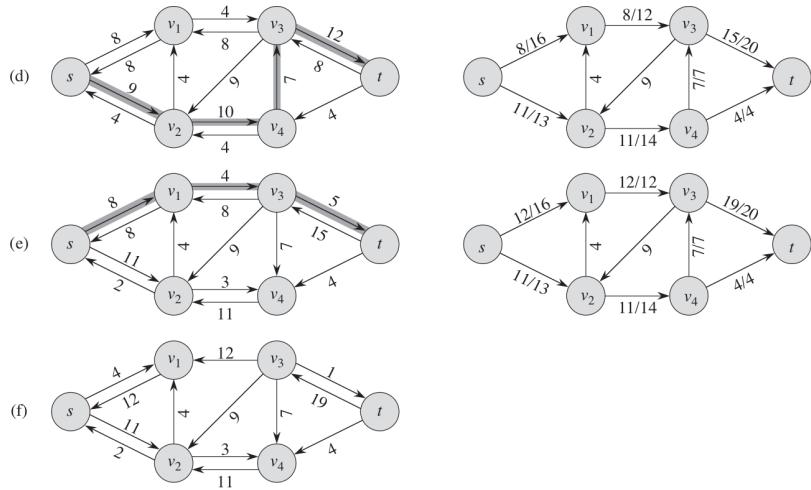


Figure 26.6, continued (f) The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

Bipartite Matching Problem

Enunciado

Uno de nuestros objetivos originales al desarrollar el Problema de flujo máximo era poder resolver el Problema de emparejamiento bipartito, y ahora mostramos cómo hacerlo. Recuerde que un grafo bipartito $G = (V, E)$ es un grafo no dirigido cuyo conjunto de nodos se puede dividir como $V = X \cup Y$, con la propiedad de que cada arista $e \in E$ tiene un extremo en X y el otro en Y . Una coincidencia M en G es un subconjunto de las aristas $M \subseteq E$ de modo que cada nodo aparece como máximo en una arista en M . El problema de coincidencia bipartita es el de encontrar una coincidencia en G del mayor tamaño posible.

Diseño de la solución

El grafo que define un problema de coincidencia no está dirigido, mientras que las redes de flujo están dirigidas; pero en realidad no es difícil usar un algoritmo para el Problema de flujo máximo para encontrar una coincidencia máxima. Comenzando con el grafo G en una instancia del problema de emparejamiento bipartito, construimos una red de flujo G' como se muestra en la figura. Primero dirigimos todas las aristas en G de X a Y . Luego

agregamos un nodo s , y una arista (s, x) de s a cada nodo en X . Agregamos un nodo t , y una arista (y, t) de cada nodo en Y a t . Finalmente, damos a cada arista en G una capacidad de 1. Ahora calculamos un flujo $s-t$ máximo en esta red G . Descubriremos que el valor de este máximo es igual al tamaño de la coincidencia máxima en G . Además, nuestro análisis mostrará cómo se puede utilizar el flujo en sí para recuperar la coincidencia.

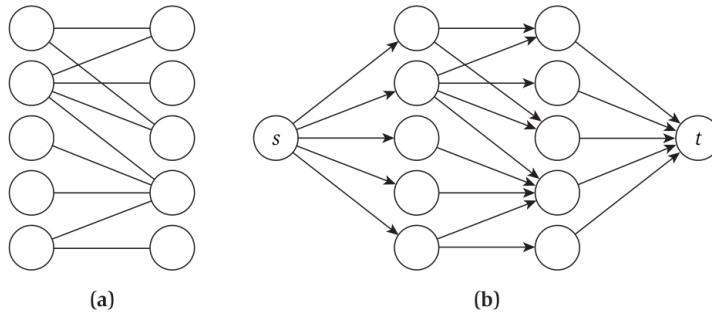


Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

Analisis de la solución

El análisis se basa en mostrar que los flujos con valores enteros en G' codifican coincidencias en G de una manera bastante transparente. Primero, suponga que hay una coincidencia en G que consta de k aristas $(x_i1, y_i1), \dots, (x_ik, y_ik)$. Luego considere el flujo f que envía una unidad a lo largo de cada camino de la forma s, x_ij, y_ij, t , es decir, $f(e) = 1$ para cada arista en uno de estos caminos. Se puede verificar fácilmente que se cumplen efectivamente las condiciones de capacidad y conservación y que f es un flujo $s-t$ de valor k .

A la inversa, suponga que hay un flujo f' en G' de valor k . Por el teorema de la integralidad para los flujos máximos, sabemos que hay un flujo f de valor entero k ; y dado que todas las capacidades son 1, esto significa que $f'(e)$ es igual a 0 o 1 para cada arista e . Ahora, considere el conjunto M' de aristas de la forma (x, y) en el que el valor de flujo es 1. Aquí hay tres hechos simples sobre el conjunto M .

- M' contiene k aristas.
- Cada nodo en X es el primero de los nodos de a lo sumo, una arista en M'
- Cada nodo en Y es el segundo de los nodos de a lo sumo, una arista en M'

Ahora consideremos qué tan rápido podemos calcular una coincidencia máxima en G . Sea $n = |X| = |Y|$, y sea m el número de aristas de G . Supondremos tácitamente que hay al menos una arista incidente en cada nodo en el problema original y, por tanto, $m \geq n/2$. El tiempo para calcular una coincidencia máxima está dominado por el tiempo para calcular un flujo máximo de valor entero en G' , ya que convertir esto en una coincidencia en G es simple. Para este problema de flujo, tenemos que $C = \sum_{(s,u)} C(s, u) = |X| = n$, ya que s tiene una arista de capacidad 1 para cada nodo de X . Por lo tanto, al usar el límite $O(mC)$ obtenemos que el algoritmo de Ford-Fulkerson usado para el problema del marching en un grafo bipartito se ejecuta en un tiempo $O(mn)$.

Diseño de encuestas

Enunciado

Un comercio de barrio desea mejorar sus ventas en base a un estudio de satisfacción de los clientes. El negocio cuenta con k productos y hasta la fecha, pasaron por sus puertas n clientes. Se necesita construir una encuesta personalizada para cada uno de estos, donde deberán opinar sobre los productos que compraron.

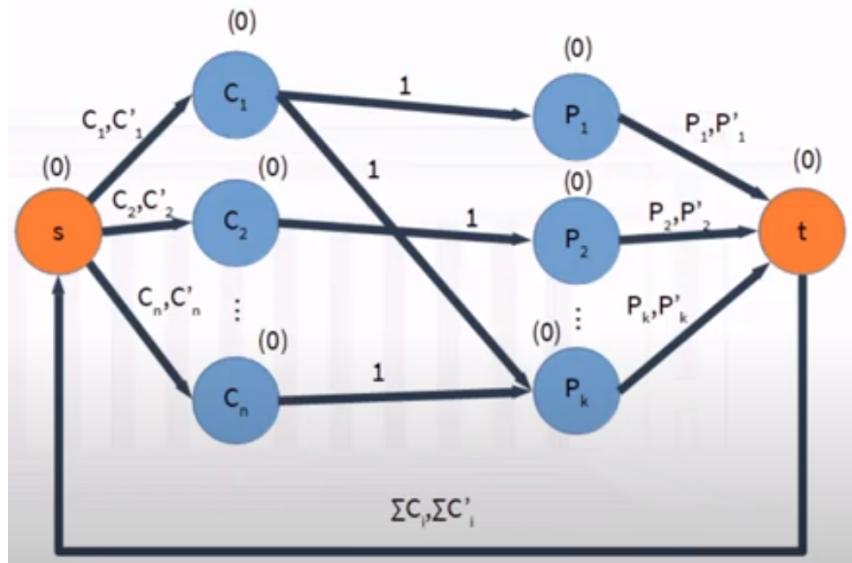
Para no saturarlos, se les pedirá a cada cliente i una cantidad de respuestas que pueden variar entre c_i y c'_i .

Por otro lado, para que se complete el estudio, se requerira que cada producto j reciba una cierta cantidad de opiniones que pueden variar entre p_j y p_j' .

Diseño de la solución

Lo primero que debemos hacer es modelar nuestro grafo respetando las restricciones propuestas. Crearemos un nodo por cliente y producto, se establecerá la relación c_i “opina sobre” el producto p_j mediante una arista dirigida. Dado que cada cliente puede opinar a lo sumo una vez sobre un mismo producto, la capacidad de la arista es igual a 1.

Hasta aqui tenemos un grafo bipartito en donde podemos decir que los nodos cliente representan fuentes de información, mientras que los nodos producto representan sumideros o receptores. Para poder continuar con la resolución del problema se debe agregar un nodo fuente s y nodo receptor t . Por cada cliente se agrega una arista con capacidades mínimas y máximas iguales a (c_i, c'_i) . Por el otro lado, se agregará una arista desde los nodos productos hacia el receptor con capacidad mínima y máxima igual a (p_j, p'_j) . Inicialmente, las demandas de cada nodo serán igual a cero. Por último, agregamos una arista (t, s) cuyo límite inferior es la suma de los límites inferiores de las aristas cliente, y con capacidad igual a la suma de las capacidades de las mismas aristas. Con esto hemos reducido nuestro problema a un problema de circulación con demanda y límites inferiores.



Ahora solo resta reducir nuestro problema a un problema de Flujo de Redes y aplicar Ford-Fulkerson. Una vez obtenido el flujo máximo, podemos decir que el $f(s, t)$ representa la cantidad total de preguntas a realizar. El flujo inyectado en las aristas (s, i) representa la cantidad de preguntas que debe responder el cliente i , mientras que el flujo presente en las aristas (j, t) son la cantidad de opiniones a recibir por cada producto.

Selección de proyectos

Enunciado

Contamos con un conjunto P de n proyectos. Cada proyecto i cuenta con un retorno económico g_i que puede ser positivo o negativo. El hecho de que sea negativo, inicialmente representa una pérdida, pero puede ser compensado con otros proyectos. Dicho sea de paso, cada proyecto puede tener a otros proyectos como requisitos para su ejecución. Queremos seleccionar un subconjunto de proyectos que maximice la ganancia.

Diseño de la solución

En primer lugar debemos hablar de un **grafo de presedencia**, que es ni más ni menos, un grafo donde se colocará un nodo por cada proyecto y donde, por cada relación del tipo “proyecto i es requisito para el proyecto j ” se colocará una arista dirigida desde p_i hasta p_j .

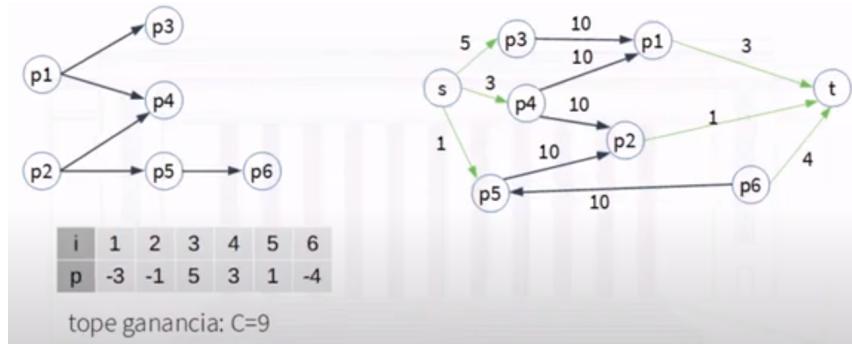
Con esto en mente podemos definir la factibilidad de los proyectos. Decimos que un subconjunto $A \subset P$ es un conjunto factible, si para cualquier proyecto $p_i \in A$ su pre requisito p_j tambien esta incluido en A.

Otro concepto necesario es el de "ganancia". Dado un subconjunto $A \subset P$ de proyectos factibles, decimos que la ganancia producida por A es $g(A) = \sum_{i \in A} gi$ Por otro lado, definimos "tope de ganancia" a la suma de todas las ganancias positivas presentes.

Con estos conceptos en mente, creamos un nuevo grafo con un nodo por cada proyecto i, un nodo fuente s y un nodo sumidero t. Por cada nodo i cuya ganancia asociada sea positiva, se crea una arista (s, i) con capacidad gi . Por cada nodo j cuya ganacia asociada sea negativa, agregamos una arista (j, t) con capacidad igual a $-gi$.

Nota: Esta asignación de aristas inicialmente nos puede parecer un poco contradictorias con lo que sabemos sobre redes de flujo con demandas. Pero podemos pensarlo de la siguiente forma. Los proyectos con ganancia positiva van a funcionar como fuente de ingreso, mientras que los proyectos con ganancia negativa, van a funcionar como demandantes de ingresos o consumidores. Desde esta perfectiva, no nos debe importar el signo de los números, sino el significado de los mismos.

Por cada arista (i, j) presente en el grafo de presedencia, agreagamos una arista (j, i) en el nuevo grafo, con una capacidad igual al tope de ganancia mas uno.



Con todo esto lo que queremos obtener es el corte minimo que lo podemos calcular con Ford-Fulkerson. Dado el corte minimo o flujo maximo, podemos afirmar que esa es la ganancia maxima a obtenerse.

Si llamamos $C(A', B')$ al corte minimos, que recordemos, esta dado por todos los nodos accesibles desde s, entonces $A' - \{s\}$ corresponde a los proyectos que se deben ejecutar para obtener la ganancia maxima.

Análisis de la solución

Sea $C(A', B')$ el corte minimo, siendo $A' = A \cup \{s\}$, $B' = (P - A) \cup \{t\}$, si p cumple con las restricciones de precedencia, el corte $C(A', B')$ no contendra aristas que unan proyectos entre A' y B' . Esto es porque la capacidad de las aristas que unen proyectos es mayor $C(\{s\}, P \cup \{t\})$ por lo que es imposible que formen parte del corte minimo.

Podemos decir que las aristas en ingresan al nodo t contribuyen al corte minimo de la siguiente forma $\sum_{i \in A / gi < 0} -gi$. Por otro lado, las aristas salientes del nodo s contribuyen al corte minimo de forma similar mediante $\sum_{i \notin A / gi > 0} gi$. Dado esto, podemos concluir que el corte minimo esta dado por

$$C(A', B') = \sum_{i \in A / gi < 0} -gi + \sum_{i \notin A / gi > 0} gi$$

Esto lo podemos reescribir con $C = \sum_{i / gi > 0} gi$ como

$$\begin{aligned} C(A', B') &= \sum_{i \in A / gi < 0} -gi + (C - \sum_{i \in A / gi > 0} gi) \\ C(A', B') &= C - \sum_{i \in A} i \in Agi \\ C(A', B') &= C - ganancia(A) \end{aligned}$$

Como C es constante, lo que el corte mínimo intentara hacer es maximizar la ganancia.

Possible ganador en torneo

Enunciado

Sea un torneo donde participan S equipos. Cada equipo s tiene una cantidad ws de partidos ganados. Para cada par de equipos x e y, les queda por enfrentarse gxy veces. Queremos determinar, dado un equipo z, si tiene posibilidad de quedar primero. Veamos un ejemplo.

	P. ganados	P. perdidos	Pend.	Atenas	Regatas	Quimsa	Peñarol
(1) Atenas	50	40	10	-	3	5	2
(2) Regatas	48	47	5	3	-	0	2
(3) Quimsa	45	48	7	5	0	-	2
(4) Peñarol	42	52	6	2	2	2	-

Algo que sabemos es que Peñarol no puede salir campeón, porque aunque gane los seis partidos que le restan, no superaría a Atenas, que ya tiene cincuenta puntos a favor. Pero que pasa con el resto?

Diseño de la solución

En principio, haremos algunos supuestos y definiciones. Si queremos saber si un equipo z puede ganar el torneo, podemos suponer que gana todos los partidos pendientes, llegando al final del torneo con $m = wz + gz$ puntos. Entonces, queremos ver si existe alguna combinación de resultados tal que otro equipo no pueda sumar más de m puntos.

En un partido entre x e y de S' , siendo $S' = S - \{z\}$, solo uno de los equipos puede ser el ganador, no existen los empates, y quien gane, se llevará un punto, mientras que el otro se llevará cero puntos.

Debemos determinar quién debe ganar los partidos para que ningún equipo supere a z. Definimos $g* = \sum_{x,y \in S'} g_{xy}$ como la cantidad de partidos por jugarse.

Ahora sí, pasemos a construir nuestra red de flujo. Creamos un grafo con un nodo vx, por cada equipo $s \in S'$. Agregamos un nodo uxy por cada par de equipos x e y en S' que tengan al menos un partido pendiente. Por último, agregamos un nodo fuente s y nodo sumidero t.

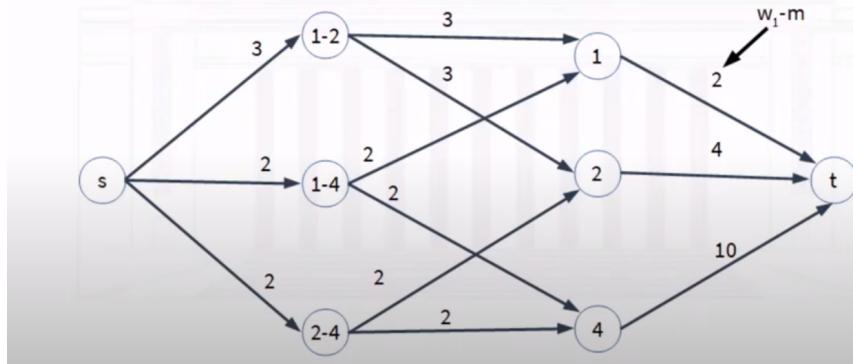
Por otro lado, creamos una arista entre:

- s y cada uxy con capacidad g_{xy} . Este representa los partidos pendientes entre x e y.
- uxy y vx con capacidad g_{xy} . Este representa los partidos entre x e y, donde x gana.
- uxy y vy con capacidad g_{xy} . Este representa los partidos entre x e y, donde y gana.
- vi y t con capacidad $m - wi$. Este representa los partidos que puede ganar i sin pasar a z.

La red de flujo de nuestro ejemplo puede verse de la siguiente forma (considerando la posibilidad de que Quimsa (3) gane todos los partidos que le restan).

Puede ganar Quimsa (3)?

Tiene 45 puntos y puede conseguir 7 más → $m = 52$



Con esto, ya podemos resolver nuestro problema mediante Ford-Fulkerson. Si existe un flujo igual a g^* , quiere decir que los puntos pueden repartirse sin que superen a los puntos del equipo z. Si el flujo es menor a g^* , esto quiere decir que z no puede quedar primero. Los puntos que aun no se repartieron, serán jugados entre los equipos, de los cuales, cualquiera puede superar a z en al menos un punto.

Segmentación de imágenes

Enunciado

Un efecto interesante que podemos lograr sobre las imágenes es el de separar el fondo del primer plano.

Sea V el conjunto de pixeles de una imagen. Podemos describir al grafo asociado a esa imagen como $G = (V, E)$, donde E es el conjunto de aristas que se trazan entre cada pixel y sus vecinos. Aunque podemos pensar a este como un grafo no dirigido, a efectos prácticos, vamos a considerarlo como un grafo dirigido donde para cada par de pixeles vecinos i y j , existen dos aristas (i, j) y (j, i) .

Entonces, para cada pixel i , tenemos un valor a_i que representara el grado en el que i pertenece al primer plano, y un valor b_i que representara el grado en el que i pertenece al fondo. Si tomamos un pixel como aislado i , podemos decir que pertenece al fondo si $a_i > b_i$.

Claro que un pixel aislado no puede por si solo definirse como parte del fondo o del primer plano. Es por esto que se propone una penalidad por cambio. Si muchos pixeles j vecinos al pixel i pertenecen al fondo, es deseable que i tambien lo sea. Entonces, por cada par de pixeles vecinos i y j , existe una penalidad p_{ij} de que pertenezcan a diferentes planos. Este valor p_{ij} es siempre mayor a cero.

Si llamamos A al conjunto de pixeles que pertenecen al primer plano y B a los pixeles que están en el fondo, entonces podemos calcular la deseabilidad de la segmentación A/B como

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{i \in A, j \in B} p_{ij}$$

Entonces, queremos seleccionar la segmentación A/B que maximice la función $q(A, B)$ o que minimice la expresión q' . Con $Q = \sum_{i \in V} a_i + b_i$ y partiendo de la igualdad $\sum_{i \in A} a_i + \sum_{j \in B} b_j = Q - \sum_{i \in A} b_i + \sum_{j \in B} a_j$.

$$q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{i \in A, j \in B} p_{ij}$$

q' quedaría definida como :

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{i \in A, j \in B} p_{ij}$$

Diseño de la solución

Con todo lo mencionado anteriormente, podemos transformar nuestro problema en un caso de corte mínimo. Para ello debemos:

- Crear un nuevo grafo G' con inicialmente los mismos V y E que G .
- Agregamos un nodo fuente s que represente al segmento fondo.
- Agregamos un nodo sumidero t que represente al segmento primer plano.
- Por cada nodo i , se creará una arista (s, i) con capacidad a_{si} .
- Por cada nodo i , se creará una arista (i, t) con capacidad b_{it} .
- Agregamos a cada arista (i, j) en E , una capacidad p_{ij} .

Y con todo esto, podemos proceder a resolver nuestro problema de corte mínimo que separe al conjunto V en dos conjuntos representantes de cada segmento.

Desde el grafo residual final, realizamos BFS desde s . Todos los nodos alcanzables serán parte del fondo, el resto pertenece al primer plano.

Programación de vuelos

Enunciado

Sea una flota de k aviones y un conjunto de m rutas de vuelos rentables, donde cada ruta está definida por un aeropuerto de inicio y otro de finalización, además de una hora de partida y otra de llegada. Deseamos determinar si podemos cubrir las rutas utilizando como mucho nuestros k aviones.

Compatibilidad de vuelos

La ruta de vuelo j es alcanzable desde la ruta de vuelo i si la ciudad de llegada de i es la ciudad de partida de j y la hora de llegada de i da tiempo de preparación suficiente a la hora de partida de j . También podemos considerar que i es compatible con j si el tiempo de vuelo y preparación desde la ciudad de llegada i a partir de su hora de llegada, es suficiente para estar en la ciudad de partida j a la hora de salida programada. Demos un ejemplo.

Contamos con

3 aviones.

Queremos cubrir

- AEP (4:30am) → MDQ (5:00am)
- AEP (8:00am) → COR (9:00am)
- IGR (5:00am) → AEP (7:00am)
- COR (11:00am) → ROS (11:30am)
- MDZ (10:00am) → AEP (11:10am)
- NQN (8:00am) → IGR (10:30am)

Vuelos compatibles:

	AEP - MDQ	AEP - COR	IGR - AEP	COR - ROS	MDZ - AEP	NQN - IGR
AEP - MDQ	SI			SI	SI	SI
AEP - COR				SI		
IGR - AEP		SI		SI		
COR - ROS						
MDZ - AEP						
NQN - IGR						

Diseño de la solución

Podemos representar a cada vuelo como:

- un nodo por cada ciudad/hora de partida
- un nodo por cada ciudad/hora de llegada
- una arista dirigida por cada ruta de vuelo

Podemos representar la compatibilidad de vuelos como

- una arista entre el nodo de llegada del vuelo i y la ciudad de partida del j

Finalmente, agregamos un nodo sumidero s y un nodo fuente t . Generamos una arista entre s y cada nodo de partida de un vuelo. Estos vienen a representar el posible inicio de recorrido de un avión. Además, agregamos una arista entre cada nodo de llegada y el nodo t . Estos vienen a representar el posible final de recorrido de un avión.

En cuanto a las capacidades, queremos que cada vuelo se ejecute una vez, por lo que ponemos una capacidad igual a uno y un límite inferior igual a uno. Esto obliga a que el vuelo se realice.

Como dijimos anteriormente, cada nodo de partida puede significar el posible inicio de recorrido de un avión, por lo que a las aristas salientes de s se asigna una capacidad de uno y un límite inferior de cero. Esto mismo se hace con las aristas entrantes en t .

Finalmente, para tratar los vuelos compatibles, se asigna una capacidad de uno y un límite inferior de cero.

Las demandas de todos los nodos es de cero a excepción del nodo s y t , donde asignamos una demanda de $-k$ y k correspondientemente. Un paso final es agregar una arista de s a t con capacidad k y límite inferior 0 que nos servirá para determinar cuantos vuelos no son necesarios.

Esto lo resolvemos mediante Ford-Fulkerson.

Complejidad NP



Casi todos los algoritmos que hemos estudiado hasta ahora han sido algoritmos de tiempo polinómico: en entradas de tamaño n , su tiempo de ejecución en el peor de los casos es $O(n^k)$ para alguna constante k . Quizás se pregunte si todos los problemas se pueden resolver en tiempo polinomial. La respuesta es no. Por ejemplo, hay problemas, como el famoso "Problema de detención" de Turing, que no puede ser resuelto por ninguna computadora, no importa cuánto tiempo le permitamos. También hay problemas que pueden resolverse, pero no en el tiempo $O(n^k)$ para cualquier constante k . En general, pensamos que los problemas que se pueden resolver mediante algoritmos de tiempo polinomial son manejables o fáciles, y los problemas que requieren tiempo superpolinomial son intratables o difíciles.

El tema de este capítulo, sin embargo, es una clase interesante de problemas, denominados problemas "NP-completos", cuyo estado se desconoce. Todavía no se ha descubierto ningún algoritmo de tiempo polinomial para un problema NP-completo, ni nadie ha podido demostrar que no pueda existir un algoritmo de tiempo polinómico para ninguno de ellos. Esta pregunta llamada $P \neq NP$ ha sido uno de los problemas de investigación abiertos más profundos y desconcertantes en la informática teórica desde que se planteó por primera vez en 1971.

Varios problemas NP-completos son particularmente tentadores porque, en la superficie, parecen ser similares a los problemas que sabemos cómo resolver en tiempo polinomial. En cada uno de los siguientes pares de problemas, uno se puede resolver en tiempo polinomial y el otro es NP-completo, pero la diferencia entre los problemas parece ser leve:

Rutas simples más cortas vs. más largas: En otros capítulos, vimos que incluso con aristas de pesos negativos, podemos encontrar las rutas más cortas de una sola fuente en un grafo dirigido $G = (V, E)$ en $O(VE)$. Sin embargo, es difícil encontrar la ruta simple más larga entre dos vértices. La mera determinación de si un grafo contiene una ruta

simple con al menos un número determinado de aristas es NP-completo.

Ciclo de Euler vs. ciclo hamiltoniano: ciclo de Euler por un grafo dirigido y conectado $G = (V, E)$ es un ciclo que atraviesa cada arista de G exactamente una vez, aunque se permite visitar cada vértice más de una vez. Podemos determinar si un grafo tiene un recorrido de Euler en solo $O(E)$, de hecho, podemos encontrar las aristas del recorrido de Euler en $O(E)$. Un ciclo hamiltoniano de un grafo dirigido $G = (V, E)$ es un ciclo simple que contiene cada vértice en V . Determinar si una grafo dirigido tiene un ciclo hamiltoniano es NP-completo. (Más adelante en este capítulo, probaremos que determinar si un grafo no dirigido tiene un ciclo hamiltoniano es NP-completo).

Completitud NP y las clases P y NP.

A lo largo de este capítulo, nos referiremos a tres clases de problemas: P, NP y NPC, siendo esta última clase los problemas NP-completos. Los describimos informalmente aquí, y los definiremos más formalmente más adelante. La clase P está formada por aquellos problemas que se pueden resolver en tiempo polinomial. Más específicamente, son problemas que pueden resolverse en el tiempo $O(n^k)$ para alguna constante k , donde n es el tamaño de la entrada al problema. La mayoría de los problemas examinados en capítulos anteriores están en P. La clase NP está formada por aquellos problemas que son "verificables" en tiempo polinomial. ¿Qué entendemos por que un problema es verifiable? Si de alguna manera nos dieran un "certificado" de una solución, entonces podríamos verificar que el certificado es correcto en el polinomio de tiempo en el tamaño de la entrada al problema. Por ejemplo, en el problema del ciclo hamiltoniano. Cualquier problema en P también está en NP, ya que si un problema está en P entonces podemos resolverlo en tiempo polinomial sin que nos entreguen un certificado. Formalizaremos esta noción más adelante en este capítulo, pero por ahora podemos creer que $P \in NP$.

De manera informal, un problema está en la clase NPC, y nos referimos a él como NP completo, si está en NP y es tan "difícil" como cualquier problema en NP. Definiremos formalmente lo que significa ser tan difícil como cualquier problema en NP más adelante en este capítulo. Mientras tanto, declararemos sin pruebas que si cualquier problema NP-completo puede resolverse en tiempo polinomial, entonces cada problema en NP tiene un algoritmo de tiempo polinomial. La mayoría de los informáticos teóricos creen que los problemas NP-completos son intratables, dado que dada la amplia gama de problemas NP-completos que se han estudiado hasta la fecha, sin que nadie haya descubierto una solución de tiempo polinomial para ninguno de ellos, sería realmente asombroso si todos ellos pudieran resolverse en tiempo polinomial. Sin embargo, dado el esfuerzo dedicado hasta ahora a demostrar que los problemas NP-completos son intratables, sin un resultado concluyente, no podemos descartar la posibilidad de que los problemas NP-completos sean de hecho solucionables en tiempo polinomial. Para convertirse en un buen diseñador de algoritmos, debe comprender los rudimentos de la teoría de la complejidad NP. Si puede establecer un problema como NP-completo, proporciona una buena evidencia de su intratabilidad. Como ingeniero, sería mejor que dedicara su tiempo a desarrollar un algoritmo de aproximación o resolver un caso especial manejable, en lugar de buscar un algoritmo rápido que resuelva el problema con exactitud. Además, muchos problemas naturales e interesantes que en la superficie no parecen más difíciles que la clasificación, la búsqueda de grafos o el flujo de red, son de hecho NP-completos. Por lo tanto, debe familiarizarse con esta notable clase de problemas.

Nos basaremos en tres conceptos claves para mostrar que un problema es NP-Completo.

Problemas de decisión vs problemas de optimización

Muchos problemas de interés son problemas de optimización, en los que cada solución factible (es decir, "legal") tiene un valor asociado, y deseamos encontrar una solución factible con el mejor valor. Por ejemplo, en un problema que llamamos SHORTEST-PATH, se nos da una grafo G no dirigido, los vértices u y v , y deseamos encontrar una ruta de u a v que use la menor cantidad de aristas. Sin embargo, la complejidad NP se aplica directamente no a los problemas de optimización, sino a los problemas de decisión, en los que la respuesta es simplemente "sí" o "no" (o, más formalmente, "1" o "0").

Aunque los problemas NP-completos se limitan al ámbito de los problemas de decisión, podemos aprovechar una relación conveniente entre los problemas de optimización y los problemas de decisión. Por lo general, podemos plantear un problema de optimización dado como un problema de decisión relacionado imponiendo un límite al valor a optimizar. Por ejemplo, un problema de decisión relacionado con SHORTEST-PATH es PATH: dado un grafo dirigido G , vértices u y v , y un entero k , ¿existe un camino desde u hasta v que conste como máximo de k aristas?

La relación entre un problema de optimización y su problema de decisión funciona a nuestro favor cuando intentamos mostrar que el problema de optimización es "difícil". Eso se debe a que el problema de la decisión es, en cierto sentido, "más fácil", o al menos "no más difícil". Como ejemplo específico, podemos resolver PATH resolviendo SHORTEST-PATH y luego comparando el número de aristas en el camino más corto encontrado con el valor del parámetro k del problema de decisión. En otras palabras, **si un problema de optimización es fácil, su problema de decisión relacionado también lo es**. Expresado de una manera que tiene más relevancia para la complejidad de NP, **si podemos proporcionar evidencia de que un problema de decisión es difícil, también brindamos evidencia de que su problema de optimización relacionado es difícil**. Por lo tanto, aunque restringe la atención a los problemas de decisión, la teoría de la complejidad de NP a menudo también tiene implicaciones para los problemas de optimización.

De aquí en mas trataremos con estos dos tipos de problemas (aunque posiblemente trabajemos mas con problemas de decisión mas que de optimización).

Reducciones.

La noción anterior de mostrar que un problema no es más difícil ni más fácil que otro se aplica incluso cuando ambos problemas son problemas de decisión. Aprovechamos esta idea en casi todas las pruebas de complejidad NP, de la siguiente manera. Consideraremos un problema de decisión A , que nos gustaría resolver en tiempo polinomial. Llamamos a la entrada a un problema particular una instancia de ese problema; por ejemplo, en PATH, una instancia sería un grafo particular G , vértices particulares u y v de G , y un entero particular k . Ahora suponga que ya sabemos cómo resolver un problema de decisión B diferente en tiempo polinomial. Finalmente, suponga que tenemos un procedimiento que transforma cualquier instancia α de A en alguna instancia β de B con las siguientes características:

- La transformación toma un tiempo polinomial.
- Las respuestas son las mismas. Esto es, la respuesta para α es "si" si y solo si la respuesta para β es tambien "si".

Nosotros llamaremos a este procedimiento un algoritmo de reducción de tiempo polinomial, que nos provee una forma de resolver problemas de A en tiempo polinomial.

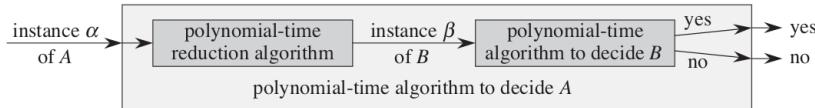


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B . In polynomial time, we transform an instance α of A into an instance β of B , we solve B in polynomial time, and we use the answer for β as the answer for α .

Siempre que cada uno de estos pasos requiera tiempo polinomial, los tres juntos también lo hacen, por lo que tenemos una manera de decidir sobre α en tiempo polinomial. En otras palabras, al "reducir" la resolución del problema A a la resolución del problema B , usamos la "facilidad" de B para demostrar la "facilidad" de A .

Recordando que NP-completo se trata de mostrar qué tan difícil es un problema en lugar de qué tan fácil es, usamos reducciones de tiempo polinomial de manera opuesta para mostrar que un problema es NP-completo. Llevemos la idea un paso más allá y mostremos cómo podemos usar reducciones de tiempo polinomial para demostrar que no puede existir un algoritmo de tiempo polinomial para un problema particular B . Suponga que

tenemos un problema de decisión A para el cual ya sabemos que no hay un algoritmo de tiempo polinomial que lo resuelva (No nos preocupemos por ahora de cómo encontrar tal problema A.) Supongamos además que tenemos una reducción de tiempo polinomial que transforma instancias de A en instancias de B. Ahora podemos usar una prueba simple por contradicción para demostrar que no puede existir un algoritmo de tiempo polinómico para B. Suponga lo contrario; es decir, suponga que B tiene un algoritmo de tiempo polinomial. Luego, utilizando el método que se muestra en la figura, tendríamos una manera de resolver el problema A en tiempo polinomial, lo que contradice nuestra suposición de que no existe un algoritmo de tiempo polinomial para A.

Un primer problema NP-completo

Debido a que la técnica de reducción se basa en tener un problema que ya se sabe que es NP-completo para probar que un problema diferente es NP-completo, necesitamos un "primer" problema NP-completo. El problema que usaremos es el problema de satisfacibilidad del circuito, en el que se nos da un circuito combinacional booleano compuesto de puertas Y, O y NO, y deseamos saber si existe algún conjunto de entradas booleanas a este circuito que provoque su la salida es 1.

Resolución en tiempo polinomial.

Para dar una mayor generalidad a los conceptos que trataremos de aqui en mas, damos dos definiciones que nos pueden ayudar.

Problemas abstractos

Para comprender la clase de problemas resolubles en tiempo polinomial, primero debemos tener una noción formal de lo que es un "problema". Definimos un problema abstracto Q como una relación binaria en un conjunto I de instancias de problemas y un conjunto S de soluciones de problemas.

Es un algoritmo, llamemoslo a, el que va a mapear estos dos conjuntos definiendo la función $a : i \in I \rightarrow S$.

Esta es un definición de muy alto nivel que nos va a servir para cualquier tipo de problema.

Caracterización de la entrada de un algoritmo

Para resolver un problema concreto con una computadora, se necesita representar dicha instancia de modo tal que un programa lo entienda y a esto es lo llamaremos parametros de un algoritmo. Los parametros de un problema computacional se terminan codificando en una cadena finita de caracteres comunmente llamada s. La longitud de $|s| = n$ es lo que usaremos despues para medir la complejidad de nuestro algoritmo.

Resolución eficiente

Por razones filosoficas y no matematicas, decimos que los problemas con soluciones polinomicas son tractables. Podemos afirmar esto por tres razones. Aunque es cierto que un solución polinomica podria ser del order $O(n^{100})$, en la practica, nunca lo son. En segundo lugar, para muchos modelos de la computación, un problema que puede ser resuelto en tiempo polinomico en un modelo, se comportara de la misma forma en otros modelos. Y por ultimo, la clase de problemas con soluciones polinomicas tienen hermosas propiedades de cierre ya que trabajan bajo operaciones de adición, multiplicación y composición. Por dar un ejemplo, si usamos la salida de un algoritmo polinomico para alimentar otro algoritmo polinomico, el proceso conjunto es polinomico.

Entonces, decimos que un algoritmo A resuelve eficientemente un problema S, si para toda instancia s de S, se encuentra la solución en tiempo polinomial.

Clase "P": Se conoce como P al conjunto de problemas de decisión para los cuales existe un algoritmo A que los resuelve de manera eficiente.

Verificación en tiempo polinomial.

Algoritmos de verificación

Suponga que un amigo le dice que un grafo G dado es hamiltoniano y luego se ofrece a probarlo dándole los vértices en orden a lo largo del ciclo hamiltoniano. Ciertamente, sería bastante fácil verificar la prueba: simplemente comprobamos si es una permutación de los vértices de V y si cada una de las aristas consecutivas a lo largo del ciclo existe realmente en el grafo. Sin duda, podría implementar este algoritmo de verificación para que se ejecute en $O(n^2)$, donde n es la longitud de la codificación de G . Por lo tanto, una prueba de que existe un ciclo hamiltoniano en un grafo se puede verificar en tiempo polinomial. Definimos un **algoritmo de verificación** como un algoritmo B de dos argumentos, donde un argumento es una cadena de entrada ordinaria s y el otro es una cadena binaria y llamada certificado. Este algoritmo B verifica una cadena de entrada s si existe un certificado y tal que $B(s, y) = 1$. Si ademas, el algoritmo B verifica en tiempo polinomial el certificado, estamos frente a un verificador eficiente.

Clase NP: Decimos que un problema de decisión es de la clase NP si existe un algoritmo de verificación eficiente.

Esta P en NP?

Sea un problema $Q \in P$, entonces existe un algoritmo A que lo resuelve de forma eficiente. Dicho esto, podemos crear un verificador B que reciba un parametro una instancia q del problema Q y un certificado t .

```
B(q, t)
  s = A(q)
  Si s == t:
    Devolver si
  Sino:
    Devolver no
```

Este verificador es eficiente porque hace uso del algoritmo A que como ya dijimos es eficiente. Entonces podemos afirmar el siguiente lema

Lema: Si $Q \in P \rightarrow Q \in NP$

Podríamos ahora, intentar probar la hipótesis contraria. Esta NP en P?. Pero lamentablemente es una pregunta abierta ya que no existe una demostración científica que corrobore o niegue a la hipótesis. Muchos conjeturan en que no, de hecho, la criptografía se basa en la hipótesis de que resolver algo es mucho mas difícil que verificar algo.

Reducciones polinomiales.

Para clasificar los problemas utilizaremos un método que se basa en transformar un problema en otro. Supongamos que tenemos una caja negra que puede resolver instancias β del problema de decisión B en tiempo polinomial. Ademas tenemos instancias α del problema de decisión A , del cual no tenemos idea de como resolver. Pues bien, si logramos reducir las instancias de A mediante un función f a instancias β del problema B , podríamos usar nuestra caja negra para resolver nuestro problema original. Claramente, necesitamos que la función f pueda realizar la reducción contraria, es decir, transformar soluciones del problema β a soluciones del problema α en tiempo polinomial. A este proceso lo llamaremos **reducción polinomial**.

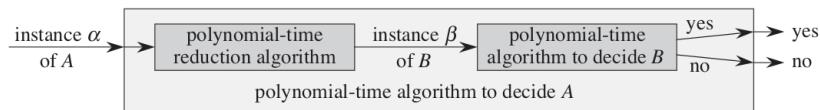


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B . In polynomial time, we transform an instance α of A into an instance β of B , we solve B in polynomial time, and we use the answer for β as the answer for α .

Si A es polinomialmente reducible a B, entonces denotaremos $A \leq_P B$. Tambien se puede decir que B es al menos "tan dificil" como resolver como A. Otra forma de verlo es la siguiente; tal vez existan formas mas eficientes de resolver A, pero en el peor de los casos, podriamos usar una reduccion polinomica a B para resolver A. Y por otro lado, dado que estamos usando una caja negra para resolver B, cuya complejidad es desconocida, solo podriamos decir que es tanto o mas dificil que resolver A.

La reducciones nos otorgan una forma de comparar a los problemas y clasificarlos si es necesario.

Lema: Sea $X \in P$ e Y un problema cualquiera. Si Y es tal que $Y \leq_P X$, entonces podemos decir que $Y \in P$.

Lema: Sea $Y \notin P$ y X un problema cualquiera. Si Y es tal que $Y \leq_P X$, entonces podemos decir que $X \notin P$.

Los lemas antes mencionados corresponden a lo que se conoce como acotar un problema a P. Algunos corolarios de estos dos lemas son:

- **Equivalencia:** Sean X e Y dos problemas tales que $Y \leq_P X$ y ademas $X \leq_P Y$. Entonces X e Y tienen la misma complejidad algoritmica.
- **Transitividad:** Sean X, Y y Z tres problemas tales que $Z \leq_P Y$ e $Y \leq_P X$, entonces $Z \leq_P X$.

Lema: Sea X un problema de desicion. X es NP-Completo si $X \in NP$ y $X' \leq_P X$ con $X' \in NP$.

Lema: Sea X un problema de desicion. X es NP-Hard si $X' \leq_P X$ con $X' \in NP$ y $X \notin NP$

Intractabilidad - co-NP

Cuando definimos un problema desicion, generalmente lo hacemos desde la afirmación, es decir, pregunamos por la existencia de una solucion. Por ejemplo, dado un grafo dirigido, existe un ciclo hamiltoniano? Dado una red de flujo, es posible transportar un flujo de F?. Ahora, uno podria trabajar con los problemas complementarios de estos. Dado un grafo dirigido, es imposible encontrar un ciclo hamiltoniano?, dado una red de flujo, es imposible encontrar un flujo de F?. Pues bien, podemos decir que para cualquier problema de desicion X, existe un problema complementario $\neg X$. Asi mismo, las instancias de X y $\neg X$ se pueden representar de la misma manera con una cade s que contiene toda la informacion necesaria para su resolucion, ejemplo, los vertices y las aristas de un grafo dirigido, o las capacidades de una red de flujo, etc. Claramente, la solucion de los problemas complementarios son opuestos, es decir, si para una instancia x de X, la respuesta es "si", entonces para su complemento $\neg X$ la respuesta sera "no".

Lema: $\neg X \in co - P$ si y solo si $X \in P$.

Lema: Sean X y $\neg X$ dos problemas complementarios uno del otro. Si $X \in P$, esto quiere decir que existe un algoritmo eficiente A para resolver cualquier instancia x de X. Entonces podemos crear un algoritmo eficiente B que resuelva $\neg X$ simplemente negando el resultado de A aplicado sobre X. Por ende $\neg X \in co - P$ y $\neg X \in P$. Entonces $P = co - P$.

Podemos ahora definir los co-NP.

Lema: Sea $X \in NP$ entonces $\neg X \in co - NP$. Otra forma de definir co-NP es como el conjunto de problemas para el cual existe un algoritmo verificador eficiente para cualquier instancia s del problema, que utilizando el certificado t de como resultado un 'no'. Es decir, el certificado es un contraejemplo del problema complementario.

La definición de co-NP-Completo es analoga a NP-Completo y lo dejamos abierto para tu verificación. Pero si podemos agregar el siguiente lema.

*Lema: Sea $X \in NP$ – Completo entonces su complemento
 $\neg X \in co-NP$ – Completo.*

Asi como la pregunta, es $P = NP$? quedo abierta. La pregunta, es $NP = co - NP$? tambien quedara sin respuesta.

Existen problemas que pertences tanto a NP como a co-NP. A estos se los conoce como problemas con buena certificacion. Todo problema que se encuentre en P tiene una buena certificación, pero mas alla de estos, se desconoces si tienen o no bueno certificacion.

NP-Completo

En este capitulo nos adentraremos en una clase de problemas, la clase NP-Completo.

Nuestros primer problema NP-Completo: SAT

Enunciado

Dado un conjunto de variables booleanas conectadas mediante operadores `or`, `and` y `not`. Se desea determinar si existe una configuracion de valores en las variables tales que el resultado de la operación sea `true`.

Analisis del problema

No existe un algoritmo eficiente para este problema o al menos aun no se encontro, por lo que una primera solucion podria ser la prueba por fuerza bruta de todas las posibles configuraciones de las variables, lo cual resulta en un orden exponencial.

Sin embargo, SAT es un problema de la clase NP. Resulta algo obvio dado que frente a un certificado, la verificacion se puede realizar en $O(n)$ siendo n la cantidad de variables.

Teorema de Cook-Levin: Sea X un problema tal que $X \in NP$, entonces $X \leq_P SAT$. Es decir, todo problema NP es a los sumo, tan complejo como SAT.

Definimos ahora la clase NP-Hard.

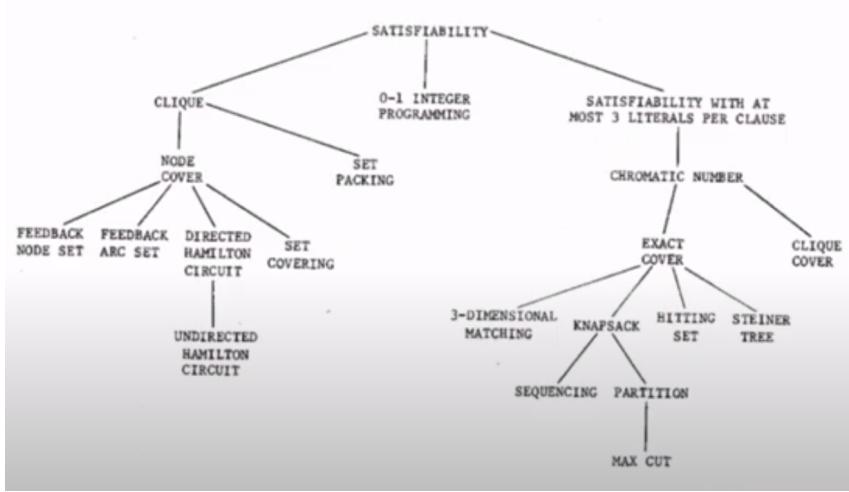
Definición: Sea X un problema tal que para todo problema $Y \in NP$ resulta que $Y \leq_P X$ entonces $X \in NP - Hard$. Esto implica que X es al menos igual de dificil que cualquier problema NP.

Definimos ahora la clase NP-Completo.

Definición: Sea X un problema tal que $X \in NP - Hard$ y $X \in NP$ entonces $X \in NP - Completo$. Esto implica que X es uno de los problemas mas difíciles dentro de NP.

Por el teorema de Cook-Levin podemos afirmar que SAT es NP-Completo, por ende SAT es el problema mas dificil entre los problemas NP. Pero esto no es todo. Sea X un problema NP, si SAT es polinomialmente reducible a X , es decir, $SAT \leq_P X$, X pasa a ser automaticamente un problema NP-Completo.

A continuacion mostramos una lista de problemas que fueron reducidos a NP-Completo.



Como se podra apreciar, los problemas mas grandes de las ciencias de la computación se puede reducir polinomialmente entre ellos. Esto quiere decir que si encontramos un algoritmo eficiente para solo uno de ellos, por transitividad, todos ellos tienen una solucion eficiente, y eso es equivalente a probar que $P = NP$.

Cobertura de vertices

Enunciado

Sea $G = (V, E)$ un grafo no dirigido. Diremos que un sub conjunto $S \subset V$ es una cobertura de vértices si para toda arista $e = (u, v)$ en E , se cumple que $u \in S$ y/o $v \in S$. Este problema tiene asociado un problema de desición que dadas las mismas condiciones, se intenta saber si existe una cobertura de vertices de tamaño al menos k .

Análisis de completitud

Recordamos que para que un problema sea NP-Completo tiene que ser NP y NP-Hard.

Es sencillo ver que la cobertura de vertices es un problema NP, dado que un verificador eficiente puede valerse de un conjunto de vertices s como certificado, y analizar con un tiempo $O(E)$ la solución. Para problema de desición asociado se debe verificar ademas que $|s| = k$.

Problemos ahora que el problema es NP-Completo. Para ello necesitamos otro problema $X \in NPC$ y reducirlo a un problema de cobertura de vertices tal que $X \leq_P CV$.

Para nuestro objetivo, usaremos el problema de los conjuntos independientes o IS que es un reconocido problema NPC.

Problema IS: Sea un grafo $G = (V, E)$ y un valor k , se desea determinar si existe un conjunto independiente de a lo sumo k vertices. Definimos un conjunto independiente $C \subset V$ como uno en el cual no existe dos pares de vértices unidos por una arista.

Podemos probar que dado un grafo $G = (V, E)$, S es un conjunto independiente si y solo si el complemento $V - S$ es una cobertura de vertices.

Primero, suponga que S es un conjunto independiente. Considere una arista arbitraria $e = (u, v)$. Como S es independiente, no puede darse el caso de que tanto u como v estén en S ; por lo que uno de ellos debe estar en $V - S$. De ello se sigue que cada arista tiene al menos un extremo en $V - S$, por lo que $V - S$ es una cubierta de vértice. A la inversa, suponga que $V - S$ es una cobertura de vértice. Considere dos nodos u y v en S . Si estuvieran unidos por la arista e , entonces ninguno de los extremos de e estaría en $V - S$, lo que contradice nuestra suposición de que $V - S$ es una cobertura de vértice. De ello se deduce que no hay dos nodos en S unidos por una arista, por lo que S es un conjunto independiente.

De esto se deduce inmediatamente que $IS \leq_P VC$. y como IS es NPC, entonces VC es NPC.

De hecho, si tenemos una caja negra para resolver la cobertura de vértice, entonces podemos decidir si G tiene un conjunto independiente de tamaño al menos k preguntando a la caja negra si G tiene una cobertura de vértice de tamaño como máximo $n - k$.

Cobertura de conjuntos

Enunciado

Sea un conjunto U de n elementos y una colección $\{s_1, s_2, \dots, s_n\}$ de subconjuntos de U . Se desea saber si existe una colección de como mucho k de los subconjuntos cuya unión es igual a U .

Análisis de completitud

Dividiremos nuestra demostración en dos partes, en la primera demostraríamos que SET-COVER es un problema NP, y en la segunda, demostraríamos que es NP-Hard, para concluir que es NP-Completo.

Un certificado para el problema SC se compondría de los subconjuntos cuya unión hace al conjunto U . Para este certificado existe un verificador eficiente que puede en $O(m + n)$ corroborar que todos los elementos de U están presentes en la unión de los subconjuntos, por tanto SC es NP.

Ahora, para demostrar que SC es NPC, haremos uso de otro problema conocido como VERTEX-COVER o cobertura de vértices.

VERTEX-COVER: Sea $G = (V, E)$ un grafo no dirigido. Diremos que un subconjunto $S \subset V$ es una cobertura de vértices si para toda arista $e = (u, v) \in E$, se cumple que $u \in S$ y/o $v \in S$. Es decir, no existe vértice que no esté conectado a S por alguna arista. Este problema tiene asociado un problema de decisión que dadas las mismas condiciones, se intenta saber si existe una cobertura de vértices de tamaño al menos k .

VERTEX-COVER es un problema NPC por lo que solo deberíamos reducirlo a SC para demostrar su completitud.

Pues bien, sea una grafo G y un valor k , construimos un conjunto tal que $E = U$, es decir, por cada arista en E , creamos un elemento en U . Por otro lado, por cada vértice $v \in V$, creamos un subconjunto s , al cual agregamos todas las aristas incidentes a ese vértice. Mantenemos el valor k como la cantidad de subconjuntos a buscar para cubrir U .

Si una caja negra pudiese resolver SC y devolver el listado de subconjuntos cuya unión forman a U , podríamos inmediatamente saber que vértices comprenden al conjunto de vértices recubridores. Por ende, $VC \leq_P SC$, concluyendo en que SC es NPC.

Clique

Enunciado

Sea una grafo $G = (V, E)$ dirigido. Llamaremos clique a un subconjunto $V' \subset V$ tal que para todo par de vértices en $u, v \in V'$ existe una arista $(u, v) \in E$. Es decir, todos los vértices en V' están conectados entre sí. Dado un valor k positivo, determinar si existe un clique de tamaño k en G .

Análisis de completitud

En este caso, un certificado estaría comprendido por los k vértices que constituyen al clique, y el algoritmo que verifique el certificado simplemente debería analizar que todos estos vértices estén conectados entre sí. Esto tomaría un tiempo $O(k^2)$ lo que lo hace un verificador eficiente, por tanto $Clique \in NP$.

3SAT Problem: Sea un conjunto de variables booleanas $X = \{x_1, x_2, x_3, \dots, x_n\}$.
 Sea un conjunto de k cláusulas booleanas del tipo $T_i = (t_{i1} \vee t_{i2} \vee t_{i3})$ con
 $t_{ij} \in X \cup -X \cup \{1\}$. Determinar si existe una asignación de variables tal que
 $T_1 \wedge T_2 \wedge \dots \wedge T_n = 1$.

Para determinar si Clique es un problema NPC usaremos 3SAT de la siguiente forma. Dada una instancia I del problema 3SAT con k cláusulas y n variables, creamos un grafo con un nodo por cada variable en una cláusula, es decir, deberíamos tener un grafo con $3 \times k$ nodos. Y por cada par de variables en diferentes cláusulas, creamos una arista que las une. Los nodos a unir no puede representar a la misma variable negada, porque son incompatibles. Buscamos un clique de tamaño k .

Si encontramos un clique de tamaño k , que es la misma cantidad de cláusulas, eso significaría que existe una asignación de variables, dado por los nodos seleccionados, tal que la instancia I resulte en uno. Los nodos que no aparezcan en el clique pueden valer cero o uno, sin afectar el resultado de la instancia I.

Esto demuestra que $3SAT \leq_P Clique$ y por tanto $Clique \in NPC$.

Subset Sum

Enunciado

Sea un conjunto $C = \{w_1, w_2, \dots, w_n\}$ de números naturales. Determinar si existe un subconjunto $s \subset C$ que sume exactamente W . Este es un problema de decisión relacionado con el problema de la mochila.

Análisis de completitud

Sea un certificado constituido por un subconjunto s de C de k elementos, existe un algoritmo eficiente que determine si la suma de esos elementos es igual a W . Por ende SS es un problema NP.

Para determinar si SS es NPC usaremos el problema 3DM.

3DM: Dados tres sets X, Y, Z de tamaño n cada uno. Se tiene un conjunto C de triples ordenadas (x, y, z) donde $x \in X, y \in Y, z \in Z$. Se desea determinar si existe un subconjunto $c \subset C$ de n triples, tal que cada elemento de la unión $X \cup Y \cup Z$ aparezca una y solo una vez en una de esas triples.

Sea una instancia I de 3DM con X, Y, Z conjuntos con n elementos y un conjunto $T \subset X \times Y \times Z$ de m triples. Podemos representar cada tripla como un vector de bits.

Utilizaremos vectores de $3 \times n$ bits. Los primeros n bits representan a los elementos de X , los siguientes n bits, representan a los elementos de Y , y los últimos n bits representan a los elementos de Z . A cada elemento del conjunto X (y similarmente para Y y Z) les asignaremos un orden arbitrario de 1 a n (recordemos que los conjuntos no tienen orden). Llamaremos $pos(x)$ a la función que dado un elemento, nos retorna su posición en el conjunto original.

Entonces, por cada tripla $t = (x, y, z) \in T$ lo representaremos como un vector de $3 \times n$ bits con todas sus posiciones en cero excepto $pos(x), pos(y) + n, pos(z) + 2n$. Con esta transformación, cada tripla queda representada por un número, que podríamos usarlo como elemento dentro de una instancia s de SS, pero tenemos un problema y es esto puede generar un overflow.

	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>...</td><td>1</td><td>0</td><td>0</td><td>...</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	0	...	1	0	0	...	0	0	1									
0	1	0	...	1	0	0	...	0	0	1											
	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>...</td><td>0</td><td>1</td><td>0</td><td>...</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	0	...	0	1	0	...	0	0	1									
0	1	0	...	0	1	0	...	0	0	1											
+	<table border="1"><tr><td></td><td></td><td></td><td>⋮</td><td></td><td></td><td></td><td>⋮</td><td></td><td></td><td>⋮</td></tr></table>				⋮				⋮			⋮									
			⋮				⋮			⋮											
	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>...</td><td>0</td><td>1</td><td>0</td><td>...</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	...	0	1	0	...	0	0	1									
1	0	0	...	0	1	0	...	0	0	1											
	<table border="1"><tr><td></td><td></td><td></td><td>⋮</td><td></td><td></td><td></td><td>⋮</td><td></td><td></td><td>⋮</td></tr></table>				⋮				⋮			⋮									
			⋮				⋮			⋮											
	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>...</td><td>0</td><td>0</td><td>1</td><td>...</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	...	0	0	1	...	1	0	0									
0	0	1	...	0	0	1	...	1	0	0											
$W =$	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>...</td><td>1</td><td>1</td><td>1</td><td>...</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	...	1	1	1	...	1	1	1									
1	1	1	...	1	1	1	...	1	1	1											

En la figura podemos ver como, intentando llegar a W , se genera un overflow que nos daria un resultado erroneo. Para subsanar esto, vamos a representar los vectores como numeros en una base d . Con esto en mente, cada tripla quedaria representada como

$$w_t = \sum_{i=1}^{3n} v[t] \times d^{i-1}$$

Y dado que casi todos los valores del vector estan en cero excepto por tres, se puede decir que

$$w_t = d^{pos(x)-1} + d^{pos(y)+n-1} + d^{pos(z)+2n-1}$$

Para la base d utilizaremos $d = m + 1$, siendo m la cantidad de triples de T . Con esto, aunque las m triples tengan a un mismo elemento, no sera posible un overflow.

Con esto, nuestra instancia s reducción de i , queda como un conjunto de valores que presentan triples y un valor W que es igual

$$W = \sum_{i=1}^{3n} d^{i-1}$$

Si tuviéramos al alcance, una caja negra que resuelve problemas del tipo SS, y este nos devolviese los elementos cuya suma es igual a W , solo restaria, retornar esos valores a su forma de triple, y obtendríamos asi una solución a 3DM. Con esto, podemos decir que $3DM \leq_P SS$, entonces $SS \in NPC$

Problema del viajante

Enunciado

Un viajante debe recorrer n ciudades v_1, v_2, \dots, v_n partiendo de v_1 , se debe construir un tour visitando cada una de las ciudades una vez y retornar a la ciudad inicial.

Para cada par de ciudades x, y se especifica una distancia $d(x, y)$. No necesariamente hay una simetria, es decir, no se cumple que $d(x, y) = d(y, x)$. Por otro lado, no necesariamente se cumple la desigualdad triangular entre las ciudades x, y, z tal que $d(x, y) + d(y, z) \geq d(x, z)$.

El problema de optimización asociado es encontrar el tour de menor distancia. Mientras que el problema de decisión asociado es determinar si existe un tour de distancia menor a k .

Análisis de completitud

Sea un certificado compuesto por un tour (listado ordenado de ciudades a visitar), se puede verificar en un tiempo $O(n)$ si la suma de las distancias entre cada par de ciudades, es menor a k . Por tanto, existe un verificador eficiente, por lo que podemos concluir con que el problema del viajante es NP.

Debemos corroborar que TP (traveler problem) es NPC. Y para eso, usaremos el problema del ciclo Hamiltoneano.

Problema del ciclo Hamiltoneano: dado un grafo $G = (V, E)$ dirigido. Se desea determinar si existe un ciclo Hamiltoneano, es decir, un tour que pasa por todos los nodos solo una vez, volviendo a su nodo inicial.

Sea I una instancia del HP, con un grafo $G = (V, E)$, por cada vértice $vi \in V$, creamos una ciudad ci . Por cada arista $e(vi, vj) \in E$ designamos una distancia $d(ci, cj) = 1$. Para cada par de nodos vi, vj para los cuales no existe una arista que los une, se designa una distancia $d(ci, cj) = 2$. Con esto obtenemos n ciudades con todas sus distancias definidas y llegamos a una instancia del HP. Si obtenemos una solución para el HP propuesto, entonces podemos transformarlo en tiempo polinomial, en una solución del TP. La reducción esta completa, por lo que concluimos que $HP \leq_P TP$ y por tanto, decimos que TP es NPC.

Nota: Existe un caso particular del problema de los ciclos hamiltonianos, que es el problema del caballo. En donde se quiere determinar si es posible recorrer todo un tablero de ajedrez, con un caballo, si pisar dos veces la misma casilla. Lo increíble es que para este problema si existe solución polinomial, a pesar de ser un caso particular de HC, lo que inmediatamente nos dice que frente a un problema, se debe analizar con cierta rigurocidad, si es o no, un caso particular de otro problema y si tiene solución.