Complejidad de algoritmos:

Problemas de grafos

Nicolas Thériault

Departamento de Matemática y Ciencia de la Computación,
Universidad de Santiago de Chile.

Estructuras de datos

En el estudio de grafos, se pueden utilizar varias estructuras de datos, cada una con sus ventajas y desventajas.

Las principales son:

- Listado de aristas
- Listado de adyacencia
- Listado dinámico
- Matriz de adyacencia

Para los problemas/algoritmos que utilizamos en esta sección del curso, utilizaremos principalmente las matrices de adyacencia.

Listado de aristas

- Se hace un listado con cada arista (a, b) en el grafo.
- Para grafos simples no dirigidos, este formato es el más compacto.
- No permite un trabajo eficiente ya que al principio no hay orden en las aristas:
  - En grafos no-dirigidos, ¿Cómo escribir la arista: (a, b) o (b, a)?
  - ¿Cómo asegurar que ninguna arista está repetida?
  - ¿Cómo determinar si hay aristas saliendo de un vértice?
  - ¿Cómo encontrar todas las aristas de un vértice?

Listado de adyacencia

- En realidad, se trata de un listado-de-listados.
- Para cada vértice, se hace un sub-listado de los vértices conectados a este primero por una arista, es decir un sub-listado de todas los "destinos" de aristas saliendo del vértice.
- Se puede empezar el sub-listado por un indice que indica cuantas entradas contiene (e.g. arreglos) o indicar por "Null" el final del sub-listado (e.g. listados de vínculos).
- Ventajas/desventajas:
  - Compacto.
  - Muy eficiente para encontrar aristas saliendo de un vértice, menos para encontrar una arista especifica (si los sub-listados son arreglos), incluso muy costoso (sub-listados de vínculos).
  - Complicado de actualizar si se eliminan o agregan aristas.

Listado (de adyacencia) dinámico

- Permite eliminar o agregar aristas rápidamente.
- Consiste en tres listados: el listado *principal*, el listado de *indices* y el listado (o stack) de *entradas no utilizadas*.
  - El listado de *indices* es un listado del número de fila de la primera arista saliendo del vértice ("Null" si no hay).
  - El listado de *entradas no utilizadas* contiene las filas del listado principal que no sirven en este instante, y se utiliza para encontrar una fila donde escribir una nueva arista.
  - El listado *principal* es una secuencia (numerada) de filas que consisten en dos entradas: el vértice final de la arista, y el número de fila de la próxima arista saliendo del vértice de origen ("Null" si la arista actual es la última).
- Ventajas/desventajas:
  - Un poco menos compacto que el listado de aristas y el listado de adyacencia (no mucho).
  - Muy eficiente para encontrar aristas saliendo de un vértice, muy costoso para encontrar una arista especifica.
  - Facíl de actualizar si se eliminan o agregan aristas.

Matriz de adyacencia

- Para representar el grafo, se utiliza una matriz $n \times n$ donde n es el número de vértices en el grafo.
- La entrada de la posición (a, b) corresponde al número de aristas que van desde el vértice a hasta el vértice b (0 si no hay).
- Ventajas/desventajas:
  - El menos compacto para grafos simples (especialmente si hay pocas aristas).
  - Muy eficiente para encontrar aristas especificas, muy costoso para determinar si hay alguna aristas saliendo de un vértice.
  - Facíl de actualizar si se eliminan o agregan aristas.

Fuerza bruta

Búsqueda En Anchura (*Breadth-First Search*):

- Datermina la distancia mínima (si existe un camino) desde un vértice orignal (fijo) a hacía otros vértices.
- También sirve a determinar todos los vértices conectados con a.
- Empezando desde a, encontrar todos los vértices directamente conectados (primer anillo), luego los vértices conectados con el primer anillo (segundo anillo), etc, hasta que se encuentra el vértice destino o no se pueda empujar más lejos.
- Requiere ordenar los vértices conectado en cada anillo.
- Complejidad general: $O(n^n)$ en tiempo y $O(n)$ en memoria.
- Complejidad en grafos de grado $\leq \omega$: $O(\omega^n)$ en tiempo (listados de adyacencia).

Fuerza bruta

Búsqueda En Profundidad (*Depth-First Search*):

- Para problemas que requieren comprobar cada camino posible.
- También para buscar caminos y ciclos de Hamilton.
- Empezando desde un vértice, intentar hacer el camino lo más largo posible.
- En cada vértice utilizado, se guarda un listado de las aristas de salida no investigados (que no re-utilizan vértices anteriores).
- Cuando no se puede seguir, hay que retroceder (*backtrack*) hasta que quedan más opciones (aristas no utilizadas).
- Complejidad general: $O(n!)$ en tiempo y $O(n^2)$ en memoria.
- Complejidad en grafos de grado $\leq \omega$: $O(\omega^n)$ en tiempo y $O(\omega n)$ en memoria.

Intercambios tiempo-memoria

Un problema importante para varias aplicaciones de grafos es de determinar, en un grafo con una cantidad de aristas creciente (en el cual se agregan aristas una por una), cuando aparece el primer ciclo y cual es este ciclo.

Por estudios teóricos, se sabe que en promedio aparece el primer ciclo cuando el grafo tiene $\approx n/2$ aristas.

Para buscar ciclos, se construyen *componentes conectados*, que son identificados por un vértice raíz (cada componente conexo es un árbol).

Cuando la cantidad de aristas se acerca a $n/2$, empieza a aparecer un *componente conexo gigante* que crece rápidamente.

Para hacerlo eficiente, se aumenta los datos (intercambio tiempo-memoria)

Intercambios tiempo-memoria

Estructuras de datos utilizados:

- listado de aristas: en el orden que se agregan al grafo (se utiliza una sola vez).
- listado de los vértices, con información suplementaria:
  - su pariente directo (si es dependiente), NULL si es aislado o raíz.
  - el peso/tamaño de su componente si es raíz (0 si es aislado, su último peso como raíz si se convertió en dependiente).

Cuando se agrega una arista, se encuentran las raíces asociados a ambos vértices:

- Si ambos lados no tienen raíz, se agrega el la arista y se utiliza uno de los dos vértices como raíz.
- Si un lado no tiene raíz (aislado), se agrega al componente de la raíz del otro vértice.
- Si hay dos raíces distintas, la arista conecta los dos componentes (les combina).
- Si las dos raíces son iguales, tenemos un ciclo.

Intercambios tiempo-memoria

Cuando se combinan dos componentes, la raíz del componente menor se convierte en dependiente de la raíz del componente mayor.

- cada vez que crece un componente, se actualiza el peso de su raíz.
- cuando se combinan dos componentes, el nuevo peso de la raíz es a lo menos el doble del peso del nuevo dependiente.

Buscar raíces se hace subiendo de pariente (raíz antigua) a pariente. Como los pesos siempre se duplican (o más), buscar la raíz es $O(\log n)$.

Complejidad: $O(n \log n)$ en tiempo y $O(n)$ en memoria.

El grafo NO se puede guardar con la matriz de adyacencia (aumentaría ambos complejidades a $O(n^2)$ solamente por crear la matriz).

Para poder extraer el ciclo, se debe agregar una información más a cada vértice: la arista que se utilizó para conectar la componente de este vértice (cuando era aislado o raíz de un componente) al componente de su pariente directo.

Programación dinámica

En vez de atacar el problema directamente, se utiliza los resultados versiones más sencilla del problema.

Observación: La programación dinámica se aplica de manera inductiva, hasta que las soluciones sean directa, luego se devuelve a la situación inicial (paso a paso).

Hay dos "clases" principales de aplicaciones de la programación dinámica, cuando se utilizan:

- Problemas de tamaño menor
  - parecido a un reducir-y-conquistar, pero donde se pueden hacer referencias a varios casos más pequeños.
- Problemas con condiciones que lo hacen más sencillo de resolver
  - Común en problemas de optimización.
  - Cuando es posible, las condiciones se combinan en una sola condición más fuerte.
  - En general, el problema con todas las condiciones (retricción máxima) se reduce a devolver la entrada del problema.

Números de Fibonacci

El n-ésimo número de Fibonacci se define por inducción como:

$$F(i) = F(i-1) + F(i-2)$$
$$F(2) = 1$$
$$F(1) = 1$$

Si se calcula por una función recursiva que primera pide $F(n-1)$ y luego (independientemente) $F(n-2)$, el algoritmo hará $\Theta(F(n))$ llamados a la función F.

Programación dinámica:

Aquí conviene de calcular todos los $F(i)$ desde $F(1)$ y $F(2)$, recordando que en cada paso solamente necesitamos los valores de $F(i-2)$ y $F(i-1)$.

Complejidad temporal: $\Theta(n)$ sumas en vez de $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.
Memoria: $\Theta(1)$ valores en vez de $\Theta(n)$.

Problema de la bajada mínima

En una pirámide (triángulo) de $n+1$ niveles, se quiere bajar del nivel n (punta) hasta el nivel 0 (planta baja).

En cada posición, hay dos opciones de escalaras (rectas) para bajar: hacía la izquierda o hacía la derecha.

Las escaleras hacen dibujos de rombos: (desde un mismo nivel) la escalera haca derecha la desde una posición y la escalera hacía la izquierda desde la posición siguiente (a la derecha de la primera), se juntan en una misma posición en el nivel siguiente.

Solamente se puede bajar.

Cada escalera tiene un costo propio (pueden ser repetidos o distintos).

En problema consiste en encontrar el camino menos costoso para bajar.

Problema de la bajada mínima

Fuerza bruta:

Comprobar cada camino posible.

En cada posición hay 2 direcciones posibles, hay que hacer n de esas elecciones.

Da un total de 2^n caminos posibles.

El costo de cada camino es la suma de n valores.

Complejidad temporal: $\Theta(n2^n)$ sumas.

Memoria: $\Theta(n)$ (para saber cual es el camino actual y/o cual es el siguiente).

Principio de los Caminos Optimales:

Teorema

En un problema de grafos, si un camino $v_1 - C - v_2$ *es un camino optimal entre los vértices* v_1 *y* v_2*, y que este camino pasa por el vértice* v_3 *de la forma siguiente:*

$$v_1 - C_1 - v_3 - C_2 - v_2$$

entonces el camino $v_1 - C_1 - v_3$ *es optimal entre* v_1 *y* v_3 *y el camino* $v_3 - C_2 - v_2$ *es optimal entre* v_3 *y* v_2*.*

Observación: No se excluye que haya más de un camino que satisface la condición de optimalidad, solamente se dice que los sub-caminos de un camino optimal también satisfacen la condición de optimalidad.

Problema de la bajada mínima

Programación dinámica:

Encontrar el costo mínimo para cada posición del k-ésimo nivel: Restricción a $k \leq n$ niveles, con inducción en k.

En el k-ésimo nivel (desde arriba, piso $n-k$), hay $k+1$ posiciones posibles.

Cada posición proviene de una (extremos) o dos (internos) posiciones del nivel anterior.

El camino menos costoso para llegar a una posición al k-ésimo nivel también era camino mínimo para su posición cuando pasó por el nivel anterior (Principio de los Caminos Optimales).

Complejidad para pasar del nivel k al nivel $k+1$: $\Theta(k)$ operaciones (sumas y comparaciones) y memoria.

Complejidad temporal (total): $\Theta(n^2)$ sumas.

Memoria: $\Theta(n)$ para encontrar el valor mínimo, $\Theta(n^2)$ para dar el camino mínimo.

Problema de la bajada mínima

Variación: el costo de la escalera viene de su posición final, no de la escalera misma.

Mismo algoritmo.

También se puede generalizar el problema:

Podemos cambiar a una pirámide en tres dimensiones, con 4 escaleras que bajan desde cada posición (bajando en dirección Norte, Este, Sur o Oeste).

La complejidad de fuerza bruta es $\Theta(n4^n)$ sumas con $\Theta(n)$ de memoria

La complejidad con programación dinámica es $\Theta(n^3)$ sumas con $\Theta(n^2)$ de memoria (para el valor mínimo, sube a $\Theta(n^3)$ para dar el camino).

Algoritmo de Floyd

Problema: Encontrar los valores de los caminos mínimos entre cada par de vértices en un grafo.

A cada arista $a_{i,j} = (v_i, v_j)$ se asocia un costo (distancia) $d_{i,j} \geq 0$.

El costo de un camino es la suma de los $d_{i,j}$ de sus aristas.

El objetivo es de encontrar el costo mínimo entre todos los caminos posibles entre los vertices v_a y v_b, y hacerlo para todos los pares a y b (idealmente al mismo tiempo).

Optimalidad: el costo mínimo entre todos los caminos posibles desde el vértices v_a y v_b.

Programación dinámica

Restricción: Considerar solamente los caminos que utilizan los k primeros vértices como vétrices intermedios.

Caso $k = 0$: grafo inicial

Caso $k = n$: todos los caminos posibles

Notación: $C^{(k)}$ es un camino que utiliza solamente los k primeros vértices como vétrices intermedios.

Principio de los Caminos Optimales:

Si un camino $v_1 - C^{(k+1)} - v_2$ es optimal, entonces o bien $C^{(k+1)} = C^{(k)}$ (optimal) o bien $C^{(k+1)} = C_1^{(k)} - v_k - C_2^{(k)}$ con ambos $C_1^{(k)}$ y $C_2^{(k)}$ optimales.

Algoritmo de Floyd

La matriz de costos $(m_{i,j})$ de las aristas del grafo se convierte en la matriz $(m_{i,j}^{(k)})$ de los costos de caminos mínimos entre v_i y v_j con vétrices intermedios $\leq k$.

Actualización del estado "k" al estado "$k+1$: Para todos los pares (i,j), elegir lo menor entre $m_{i,j}^{(k-1)}$ y $m_{i,k}^{(k-1)} + m_{k,j}^{(k-1)}$

Algoritmo:

$(m_{i,j}) \leftarrow$ matriz de adyacencia

Para k desde 1 hasta n:

Para i desde 1 hasta n, $i \neq k$:

Para j desde 1 hasta n, $j \neq k$:

$m_{i,j} \leftarrow \min\{m_{i,j}, m_{i,k} + m_{k,j}\}$

Devolver $(m_{i,j})$.

Complejidad: tiempo $\Theta(n^3)$, memoria $\Theta(n^2)$.

Algoritmo de Warshall

Problema: Determinar cuales pares de vértices de un grafo están conectados.

Visión aditiva: Se puede utilizar el algoritmo de Floyd donde los costos de las aristas es 0 (arista presente), entendiendo que ∞ corresponde a una ausencia de conexión.

Visión aditiva: Utiliza lógica con 1 (presente) y 0 (ausente).

Las dos visiones son equivalentes:

$$\begin{array}{ccc} a = 0 & & \tilde{a} = 1 \\ a = \infty & & \tilde{a} = 0 \\ a + b & \longleftrightarrow & \tilde{a} \cdot \tilde{b} \\ & & \tilde{a} \wedge \tilde{b} \\ \min\{a, b\} & & \tilde{a} \vee \tilde{b} \end{array}$$

Algoritmo de Warshall

La matriz de adyacencia $(m_{i,j})$ de las aristas del grafo se convierte en la matriz $(m_{i,j}^{(k)})$ de conectividad por caminos entre v_i y v_j con vétrices intermedios $\leq k$.

Actualización del estado "k" al estado "$k+1$: Para todos los pares (i,j), ver si $m_{i,j}^{(k-1)}$ o $m_{i,k}^{(k-1)} \times m_{k,j}^{(k-1)}$ da 1

Algoritmo:

$(m_{i,j}) \leftarrow$ matriz de adyacencia

Para k desde 1 hasta n:

Para i desde 1 hasta n, $i \neq k$:

Para j desde 1 hasta n, $j \neq k$:

$m_{i,j} \leftarrow m_{i,j} \vee (m_{i,k} \wedge m_{k,j})$

Devolver $(m_{i,j})$.

Complejidad: tiempo $\Theta(n^3)$, memoria $\Theta(n^2)$.

Camino mínimo

Aunque la matriz obtenida por el algoritmo de Floyd da solamente el valor del camino mínimo, es posible utilizarla para extraer los pasos de un camino mínimo entre a y b en tiempo $O(n^2)$, con camino dado como listado de vínculos:

$j \leftarrow 1$, $C[0] \leftarrow a$, $C[1] \leftarrow b$
Para k desde n bajando hasta 1, $k \neq a, b$:
    Para i desde 1 hasta j:
        Si $m_{C[i-1],k} + m_{k,C[i]} = m_{C[i-1],C[i]}$:
            Insertar k entre $C[i-1]$ y $C[i]$, $j \leftarrow j+1$, pasar al siguiente k
Devolver C.

Observación: se puede hacerlo más rápido (tiempo $O(n \log n)$) si guardamos todas las matrices $m_{i,j}^{(k)}$ en el algoritmo de Floyd, pero el costo en memoria aumenta a $\Theta(n^3)$:

$i \leftarrow 0$, $C[0] \leftarrow a$, $C[1] \leftarrow b$
Mientrás $C[i] \neq b$:
    Encontrar el menor k tal que $m^{(k)}_{C[i],C[i+1]} = m^{(n)}_{C[i],C[i+1]}$:
        Si $k > 0$:
            Insertar k entre $C[i]$ y $C[i+1]$
        Si no, $i \leftarrow i+1$
Devolver C.

Técnicas de desarrollo

Algoritmo codicioso:

Idea: Reducir el problema tomando la decisión que parece la más ventajosa en el momento, sin preocupación para el futuro.

Observación: La técnica tiene sentido solamente para problemas de optimización.

Se construye una solución (no necesariamente la optimal) haciendo pequeñas decisiones que son:

- posibles (que no contradicen el problema)
- localmente optimales (parecen las mejores opciones en este momento)
- irrevocables (no se puede reconsiderar o cancelar un decisión)

Algoritmos codiciosos y optimalidad

- En general, los algoritmos codiciosos solamente dan una aproximación de la solución optimal.
- Analizar cuando cercano a optimal el resultado realmente es puede ser un trabajo difícil.
- En algunos casos, se puede demostrar que el resultado del algoritmo codicioso es optimal, pero no es tan común.
- El principal interés de los algoritmos codiciosos es de permitir de llegar a soluciones razonablemente buena de problemas intratables.

Ejemplo: el algoritmo LLL para encontrar el vector ($\neq 0$) más corto en un reticulado.

A pesar de no siempre dar resultados optimales, el algoritmo LLL permite resolver (de manera exacta) varios problemas, e incluso sirve a demostrar algunos teoremas.

Árbol generador mínimo

Problema: En un grafo conexo con pesos (aristas con valores), encontrar un árbol que conecta todos los vértices y tiene peso total mínimo.

Este problema tiene aplicaciones naturales en comunicaciones, redes de distribución, etc.

Condición de optimalidad:

- El peso de un árbol es la suma de los pesos de todas sus aristas.
- Cuando se trabaja con sub-árboles , se considera la optimalidad en el sub-grafo que corresponde a los vértices del sub-árbol.

Con esta definición, podemos aplicar el equivalente del Principio de los Caminos Optimales (aquí deberíamos decir el **Principio de los Árboles Optimales**).

Algoritmo de Prim

Idea: Agregar los vértices uno por uno, siempre agregando con la arista la más barata possible.

Necesitamos un listado de todos los vértices no conectados al árbol actual, con el costo mínimo para conectarlos (valor de la arista) y donde se conectarían (segundo vértice de la arista).

Este listado se debe actualizar cada vez que

Si representamos el grafo con una matriz de costos, la complejidad total es $O(n^2)$.

Si representamos el grafo con un listado de adyacencia (con costos), se puede reducir la complejidad a $O(m \log n)$ donde m es la cantidad de aristas.

Para eso, debemos trabajar con un montículo (dando prioridad al valor mínimo), que se debe actualizar cada vez que una arista puede empezar a ser considerada (cuando uno de sus vértices se agrega al grafo.

Optimalidad

Para demostrar que el algoritmo de Prim da un resultado optimal, lo hacemos por inducción.

- Por el Principio de los Árboles Optimales, todo sub-árbol del árbol final debe satisfacer la condición de optimalidad también.
- Supongamos que el sub-árbol con las primeras k aristas es un sub-árbol optimal.
  - Para $k = 0$ (solamente el primer vértice), eso es claro.
  - Es decir, existe un árbol optimal que incluye este sub-árbol A_k.
- Supongamos que el sub-árbol con las primeras $k + 1$ aristas ya no es sub-árbol optimal.
  - Es decir, ningún árbol optimal $\tilde{A}$ (que incluye el sub-árbol A_k) puede utilizar la $k + 1$-ésima arista (a, b).
  - Si agregamos la $k + 1$-ésima arista al árbol optimal $\tilde{A}$, tendríamos un ciclo.
  - Sea $(\tilde{a}, \tilde{b})$ la otra arista del ciclo que conecta el sub-árbol A_k con la parte de $\tilde{A}$ que no es de A_k.
  - intercambiar $(\tilde{a}, \tilde{b})$ con (a, b) en $\tilde{A}$ da un nuevo árbol A, que por construcción de (a, b) debe tener costo menor o igual al costo de $\tilde{A}$.
  - por lo tanto o bien $\tilde{A}$ no era optimal, o bien ambos A y $\tilde{A}$ son optimales.
  - Entonces existe un árbol optimal con A_k y (a, b).
- Por inducción, llegamos que el árbol final (A_{n-1}) debe ser optimal.