Técnicas fundamentales de diseño de algoritmos (Algoritmos codiciosos)

IIC2283 – Diseño y Análisis de Algoritmos

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

2025-2

Algoritmos codiciosos

Los algoritmos codiciosos (greedy en inglés) son usados, en general, para resolver problemas de optimización.

El principio fundamental de este tipo de algoritmos es lograr un óptimo global tomando decisiones locales que son óptimas.

- En general, la intuición detrás de estos algoritmos es simple

Dos componentes fundamentales de un algoritmo codicioso:

- Una función objetivo a minimizar o maximizar
- Una función de selección usada para tomar decisiones locales óptimas

Algoritmos codiciosos

- Los problemas de optimización generalmente se resuelven mediante algoritmos que toman decisiones a lo largo de una secuencia de pasos.
- Los algoritmos **codiciosos** construyen la solución a un problema mediante una secuencia de pasos.
  - Cada paso expande la solución parcial construida hasta el momento, hasta llegar a la solución completa.
- En cada paso, el algoritmo codicioso toma una decisión que debe cumplir con las siguientes propiedades:
  1. **Factible**: debe satisfacer las restricciones al problema.
  2. **Localmente óptima**: debe ser la mejor elección entre todas las elecciones factibles.
  3. **Irrevocable**: una vez tomada una decisión, no puede ser cambiada en los siguientes pasos del algoritmo.

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

- Sin embargo, en muchos casos pueden ser usados como heurísticas para encontrar valores de la función objetivo cercanos al óptimo

Una segunda dificultad con los algoritmos codiciosos es que, en general, se necesita de una demostración formal para asegurar que encuentran el óptimo global

- Aunque pueden ser algoritmos simples, en algunos casos no es obvio por qué encuentran el óptimo global
- La dificultad está en la demostración de optimalidad, más que en el algoritmo

Algoritmos codiciosos

Algunos ejemplos clásicos de algoritmos codiciosos:

- El algoritmo de Dijkstra para caminos más cortos en grafos
- Los algoritmos de Prim y Kruskal para minimum spanning trees
- Algoritmo de Huffman para codificar fuentes de datos
- Algoritmo de coloreo de grafos
- Algoritmo para planificación de tareas
- etc.

Estudiaremos algunos ejemplos clásicos de algoritmos codiciosos

Algoritmos codiciosos: la fila de monedas

- Consideremos el problema de recoger n monedas dispuestas sobre una fila sobre la mesa.
- El problema consiste en tomar la mayor cantidad de dinero de la mesa.
- Sujeto a la restricción de cada vez que se tome una moneda, quedan invalidadas las dos monedas que estaban contiguas a la moneda escogida.
- Por ejemplo, consideremos la siguiente fila de monedas:

$$1, 5, 10, 1, 5, 10, 50, 10, 1, 1$$

Algoritmos codiciosos: la fila de monedas

- Una estrategia greedy toma, sin dudas, la moneda de mayor denominación, en este caso la de $\$50$

$$1, 5, 10, 1, 5, \underbrace{10}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{10}_{\times}, 1, 1$$

- A continuación, la estrategia greedy escoge la tercera moneda en la fila (de $\$10$).
- Luego se escoge la quinta moneda de la fila (de $\$5$)
- Finalmente, en los últimos dos pasos escogen las monedas de $\$1$

$$\underbrace{1}_{\checkmark}, \underbrace{5}_{\times}, \underbrace{10}_{\checkmark}, \underbrace{1}_{\times}, \underbrace{5}_{\checkmark}, \underbrace{10}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{10}_{\times}, \underbrace{1}_{\times}, \underbrace{1}_{\checkmark}$$

- Total recogido $\$67 \longrightarrow$ en este caso coincide con la cantidad óptima para esa fila

Algoritmos codiciosos: la fila de monedas

- Eso podría hacernos sospechar que esta estrategia greedy, después de todo, es óptima.
- Sin embargo, no lo es en general.
- El hecho de tomar siempre la moneda de mayor denominación, sin considerar el valor de las monedas contiguas que está invalidando, lleva a obtener soluciones **sub óptimas**.
- El siguiente ejemplo ilustra esto:

$$\underbrace{25}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{26}_{\times}$$

- La estrategia escoge la moneda de $\$50$ e invalida las otras dos, siendo que estas suman $51 > 50$.
- La solución correcta a este problema puede encontrarse con programación dinámica, en tiempo y espacio $O(n)$.

Algoritmos codiciosos: división de un texto en líneas

- Sea

$$T = p_1, p_2, \ldots, p_n$$

un texto formado por n palabras p_i $(1 \leqslant i \leqslant n)$, de longitudes

$$l_1, l_2, \ldots, l_n$$

- El problema consiste en imprimir el texto en la pantalla (o en una hoja de papel), teniendo en cuenta:
  - el ancho de línea predefinido es L (no se puede cambiar)
  - las palabras no pueden cortarse para ajustarlas a una línea
  - la amplitud ideal de los de los espacios entre palabras es b
    - pueden ampliarse o reducirse, para acomodar las palabras en una línea.
- En resumen, se quiere dividir el texto en líneas para su impresión

Algoritmos codiciosos: división de un texto en líneas

- Considere la secuencia de palabras contiguas $p_i, p_{i+1}, \ldots, p_j$, la cual es separada por $j - i$ espacios en blanco
- Si en un línea modificamos el valor b por un valor alternativo b', se obtiene una penalización de

$$(j - i)|b' - b|$$

puntos

- Si queremos colocar la secuencia $p_i, p_{i+1}, \ldots, p_j$ en una línea, entonces

$$b' = \frac{L - l_i - l_{i+1} - \cdots - l_j}{j - i}$$

- El objetivo es dividir el texto en líneas de manera que la penalización total sea mínima.

Algoritmos codiciosos: división de un texto en líneas

Estudiemos una solución codiciosa para este problema

- La idea es comenzar agregando las palabras $p_1, p_2, \ldots$ en la línea actual usando el espacio ideal de tamaño b, hasta que una palabra p_a no quepa en la línea actual.
- Es decir,
$$l_1 + l_2 + \cdots + l_a + (a-1)b > L$$
- En este punto se debe tomar una decisión:
  1. Imprimir p_a en la línea actual, reduciendo la amplitud de los espacios en blanco.
  2. Imprimir p_a en la línea siguiente, incrementando la amplitud de los espacios en blanco.
- La solución codiciosa elige la alternativa que produce la menor penalización (en la línea actual).
- Aunque esta solución es eficiente ($T(n) \in \Theta(n)$), produce soluciones sub óptimas.

Algoritmos codiciosos: división de un texto en líneas

Por ejemplo, sean:

- $L = 26$
- $b = 2$
- $n = 7$
- $l_1, \ldots, l_7 = 10, 10, 4, 8, 10, 12, 12$

En ese caso, el algoritmo codicioso produce una división $(10, 10, 4)(8, 10)(12, 12)$, con penalización total $2 + 6 + 0 = 8$

Una división de la forma $(10, 10)(4, 8, 10)(12, 12)$ produciría una penalización menor: $4 + 0 + 0 = 4$

Nuevamente, el problema puede resolverse de forma correcta usando programación dinámica.

Algoritmos codiciosos

Los dos ejemplos anteriores corresponden a algoritmos codiciosos que resuelven problemas de optimización de forma no correcta:

- No se puede demostrar que encuentran siempre la solución óptima

En este curso, estaremos interesados en estrategias codiciosas para las que se pueda demostrar correctitud

Algoritmos codiciosos: planificación de tareas

- Suponga un servidor que debe atender n tareas, cuyos tiempos requeridos para completarse son $t_1, \ldots, t_n$
- El servidor sólo puede atender una tarea a la vez
- Una vez que comenzó a resolver una tarea, ésta no puede ser abandonada hasta su finalización
- Las tareas están en una cola esperando a ser atendidas
- El servidor podría atenderlas en orden de llegada, o bien podría planificar en qué orden las atiende
- El problema es encontrar en qué orden ejecutar las tares de manera que la suma total de los tiempos de espera sea el mínimo posible
  - Hay $n!$ distintos órdenes

Algoritmos codiciosos: planificación de tareas

- Suponga 3 tareas con tiempos requeridos $t_1 = 4$, $t_2 = 6$, $t_3 = 2$
- Si el servidor las ejecutara en ese orden, la primera tarea no tiene que esperar nada para ser realizada, completándose su servicio en 4 unidades de tiempo.
- La segunda tarea esperará 4 unidades mientras se ejecuta la primera tarea, completándose en 4+6=10 unidades de tiempo.
- La última tarea espera 10 unidades de tiempo hasta comenzar a ser resuelta, y finaliza en el tiempo 12.
- El total de tiempos de espera es de $0 + 4 + 10 = 14$ unidades de tiempo.
- Note que si ejecutamos la tarea 3 inicialmente, luego la tarea 1, y finalmente la tarea 2, el tiempo total de espera sería:

$$0 + 2 + (2 + 4) = 2 + 6 = 8$$

Algoritmos codiciosos: planificación de tareas

- Diseñaremos un algoritmo codicioso para resolver este problema
- Sea $i_1, i_2, \ldots, i_n$ una permutación de $[1, n]$ que indica el orden de ejecución de las tareas
- Ejemplo: para 5 tareas una permutación podría ser $3, 4, 2, 5, 1$: primero la tarea 3, luego 4, luego 2, seguido de 5, y finalmente 1
- Note que los tiempos de espera de cada tarea son:
  - Tarea i_1: 0
  - Tarea i_2: $0 + t_{i_1}$
  - Tarea i_3: $0 + t_{i_1} + t_{i_2}$
  - Tarea i_4: $0 + t_{i_1} + t_{i_2} + t_{i_3}$
  - $\vdots$
  - Tarea i_n: $0 + t_{i_1} + t_{i_2} + \cdots + t_{i_{n-1}}$
- Resumiendo, el tiempo de espera total es:

$$\sum_{j=1}^{n} \sum_{k=1}^{j-1} t_{i_k}$$

Algoritmos codiciosos: planificación de tareas

- La idea es encontrar una permutación $i_1, \ldots, i_n$ de $[1..n]$ tal que la suma:

$$\sum_{j=1}^{n} \sum_{k=1}^{j-1} t_{i_k}$$

  sea mínima.

- Esa es la función objetivo del algoritmo, la cual debe ser minimizada
- Note que:

$$\begin{aligned} \sum_{j=1}^{n} \sum_{k=1}^{j-1} t_{i_k} &= (n-1) \cdot t_{i_1} + (n-2) \cdot t_{i_2} + \cdots + 1 \cdot t_{i_{n-1}} \\ &= \sum_{j=1}^{n} (n-j) t_{i_j} \end{aligned}$$

Algoritmos codiciosos: planificación de tareas

- El término t_{i_1} es el que más se suma ($n-1$ veces)
  - Si queremos minimizar la suma, t_{i_1} debería ser la menor de todas las duraciones
- El término t_{i_2} es el segundo que más se suma ($n-2$ veces), por lo que t_{i_2} debería ser el segundo menor valor si queremos minimizar la suma
- Y así siguiendo, lo cual sugiere nuestro algoritmo greedy:
  - Para minimizar el tiempo total de espera, las tareas deben ejecutarse en orden creciente de sus duraciones, desde la tarea más corta a la más larga
  - En cada paso, el algoritmo greedy decide ejecutar la siguiente tarea más corta disponible a ser ejecutada
- Este algoritmo permite encontrar el valor óptimo buscado

Algoritmos codiciosos: codificación de una fuente de datos

- Estudiamos a continuación el problema de codificar una fuente que genera secuencias de símbolos (strings)
- Suponga una fuente aleatoria que genera símbolos de un alfabeto Σ tal que para todo $a \in \Sigma$, $\Pr(a)$ es la probabilidad de que el símbolo a sea generado por la fuente
- La fuente genera secuencias de símbolos (llamados textos o strings) con esa distribución de probabilidad
- La secuencia de símbolos se enviará a través de un canal de comunicación a un destino
- Aplicaciones:
  - Comunicación de datos en una red
  - Envío de datos desde un sensor
  - Aplicaciones de mensajería
  - Telegrafía
  - etc.

Algoritmos codiciosos: codificación de una fuente de datos

- Estudiamos a continuación el problema de codificar una fuente que genera secuencias de símbolos (strings)
- Suponga una fuente aleatoria que genera símbolos de un alfabeto Σ tal que para todo $a \in \Sigma$, $\Pr(a)$ es la probabilidad de que el símbolo a sea generado por la fuente
- La fuente genera secuencias de símbolos (llamados textos o strings) con esa distribución de probabilidad
- La secuencia de símbolos se enviará a través de un canal de comunicación a un destino
- Aplicaciones:
  - Comunicación de datos en una red
  - Envío de datos desde un sensor
  - Aplicaciones de mensajería
  - Telegrafía
  - etc.

Algoritmos codiciosos: codificación de una fuente de datos

- En el destino, la secuencia debería poder leerse para entender qué datos fueron enviados
- El problema consiste en asignarle un código que permita identificar a cada uno de los símbolos generados por la fuente, para que puedan ser decodificados
- El problema se conoce como *codificación de la fuente* (source coding problem)
- Uno quisiera transmitir la menor cantidad de bits posible, por lo que buscamos codificaciones eficientes (que usen pocos bits)

Algoritmos codiciosos: codificación de una fuente de datos

Dado un string w de símbolos de un alfabeto Σ, suponga que queremos codificar w utilizando los símbolos 0 y 1

- Esta palabra puede ser pensada como un texto si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga
- El uso de 0 y 1 corresponde a la idea de codificar/almacenar el texto en un computador

Definimos entonces una función $\mathcal{C} : \Sigma \mapsto \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ un código $\mathcal{C}(a) \in \{0,1\}^*$ con $\mathcal{C}(a) \neq \varepsilon$

- Llamamos a $\mathcal{C}$ una Σ-codificación
- Por ejemplo, para $\Sigma = \{a, b, c, d\}$ se podría definir $\mathcal{C}$ tal que $\mathcal{C}(a) = 11$, $\mathcal{C}(b) = 10$, $\mathcal{C}(c) = 110$, $\mathcal{C}(d) = 111$
- Las Σ-codificaciones deberán cumplir ciertas propiedades para ser útiles

Algoritmos codiciosos: codificación de una fuente de datos

- Una Σ-codificación $\mathcal{C} : \Sigma \mapsto \{0,1\}^*$ es *no singular* si

$$\forall x, x' \in \Sigma,\ x \neq x' \implies C(x) \neq C(x')$$

- Esto es, cada elemento de Σ es mapeado a un string distinto de $\{0,1\}^*$

- La no singularidad es suficiente para una descripción no ambigua de un único símbolo de Σ

- Sin embargo, queremos codificar un string (secuencia de símbolos) de forma no ambigua, y la no singularidad no es suficiente

Algoritmos codiciosos: codificación de una fuente de datos

- La extensión $\mathcal{C}^*$ de una Σ-codificación $\mathcal{C}$ a todas las palabras $w \in \Sigma^*$ se define como:

$$\mathcal{C}^*(w) \quad = \quad \begin{cases} \varepsilon & w = \varepsilon \\ \mathcal{C}(a_1) \cdots \mathcal{C}(a_\ell) & w = a_1 a_2 \cdots a_\ell \text{ con } \ell \geqslant 1 \end{cases}$$

- Una Σ-codificación $\mathcal{C}$ es unívocamente decodificable si su extensión $\mathcal{C}^*$ es no singular

- Esto es, $\forall w_1, w_2 \in \Sigma^*,\ w_1 \neq w_2 \implies \mathcal{C}^*(w_1) \neq \mathcal{C}^*(w_2)$

Algoritmos codiciosos: codificación de una fuente de datos

- Por ejemplo, para $\Sigma = \{a, b, c, d\}$ la codificación $\mathcal{C}(a) = 11$, $\mathcal{C}(b) = 10$, $\mathcal{C}(c) = 110$, $\mathcal{C}(d) = 111$, no es unívocamente decodificable, ya que

$$db \neq ac \text{ y } \mathcal{C}^*(db) = \mathcal{C}^*(ac)$$

- Significa que si la fuente codifica 11110, en el destino no sabremos si corresponde al string db o a ac
- Note que una Σ-codificación $\mathcal{C}$ es unívocamente decodificable si su extensión $\mathcal{C}^*$ es una función inyectiva
- ¿Cómo podemos lograr esta propiedad?

Algoritmos codiciosos: codificación de una fuente de datos

Codificaciones de largo fijo

- Una manera sencilla de obtener una codificación $\mathcal{C}$ unívocamente decodificable es imponiendo la siguiente condición:

  para cada $a, b \in \Sigma$ se tiene que $|\mathcal{C}(a)| = |\mathcal{C}(b)|$

- Decimos entonces que $\mathcal{C}$ es una Σ-codificación de largo fijo

Ejercicio

Muestre que para tener una Σ-codificación de largo fijo basta con asignar a cada $a \in \Sigma$ un código $\mathcal{C}(a) \in \{0,1\}^*$ tal que $|\mathcal{C}(a)| = \lceil \log_2(|\Sigma|) \rceil$

Codificaciones de largo variable

Decimos que $\mathcal{C}$ es una Σ-codificación de largo variable si existen $a, b \in \Sigma$ tales que $|\mathcal{C}(a)| \neq |\mathcal{C}(b)|$

¿Por qué nos conviene utilizar codificaciones de largo variable?

- Podemos obtener representaciones más cortas para la palabra que queremos almacenar

Codificaciones de largo variable

Ejemplo

Suponga que $w = aabaacaab$

- Para $\mathcal{C}_1(a) = 00$, $\mathcal{C}_1(b) = 01$ y $\mathcal{C}_1(c) = 10$ tenemos que:

$$\mathcal{C}_1^*(w) \quad = \quad 0000010000010000001$$

- Para $\mathcal{C}_2(a) = 0$, $\mathcal{C}_2(b) = 10$ y $\mathcal{C}_2(c) = 11$ tenemos que:

$$\mathcal{C}_2^*(w) \quad = \quad 001000110010$$

Por lo tanto $|\mathcal{C}_2^*(w)| = 12 < 18 = |\mathcal{C}_1^*(w)|$

Pero nos falta responder una pregunta: ¿por qué $\mathcal{C}_2^*$ es inyectiva?

Codificaciones de largo variable

Lema

Sea $\mathcal{C}$ una Σ-codificación no singular. Si existen $w_1, w_2 \in \Sigma^*$ tales que $w_1 \neq w_2$ y $\mathcal{C}^*(w_1) = \mathcal{C}^*(w_2)$, entonces existen $a, b \in \Sigma$ tales que $a \neq b$ y $\mathcal{C}(a)$ es un prefijo de $\mathcal{C}(b)$

Demostración: Suponga que $w_1 \neq w_2$, $\mathcal{C}^*(w_1) = \mathcal{C}^*(w_2)$ y

$$\begin{array}{lcll} w_1 & = & a_1 \dots a_m & m \geqslant 1 \\ w_2 & = & b_1 \dots b_n & n \geqslant 1 \end{array}$$

Además, sin perdida de generalidad suponga que $m \leqslant n$

Si w_1 es prefijo propio de w_2 entonces $\mathcal{C}^*(w_1)$ es prefijo propio de $\mathcal{C}^*(w_2)$

- Puesto que $\mathcal{C}(a) \neq \varepsilon$ para cada $a \in \Sigma$

Dado que $\mathcal{C}^*(w_1) = \mathcal{C}^*(w_2)$, tenemos entonces que w_1 no es prefijo propio de w_2

Codificaciones de largo variable

Sea $k = \text{mín}\{i \mid 1 \leqslant i \leqslant m \ \wedge \ a_i \neq b_i\}$

- k está bien definido puesto que $w_1 \neq w_2$ y w_1 no es prefijo propio de w_2

Dado que $\mathcal{C}^*(a_1 \cdots a_{k-1}) = \mathcal{C}^*(b_1 \cdots b_{k-1})$ y $\mathcal{C}^*(w_1) = \mathcal{C}^*(w_2)$, concluimos que $\mathcal{C}^*(a_k \cdots a_m) = \mathcal{C}^*(b_k \cdots b_n)$

Tenemos entonces que $\mathcal{C}(a_k) \cdots \mathcal{C}(a_m) = \mathcal{C}(b_k) \cdots \mathcal{C}(b_n)$ y $\mathcal{C}(a_k) \neq \mathcal{C}(b_k)$ (porque $\mathcal{C}$ es no singular), de lo cual se deduce que $\mathcal{C}(a_k)$ es un prefijo de $\mathcal{C}(b_k)$ o $\mathcal{C}(b_k)$ es un prefijo de $\mathcal{C}(a_k)$

- Lo cual concluye la demostración puesto que $a_k \neq b_k$ $\square$

Codificaciones libres de prefijos

Decimos que una Σ-codificación $\mathcal{C}$ es libre de prefijos si para cada $a, b \in \Sigma$ tales que $a \neq b$ se tiene que $\mathcal{C}(a)$ no es un prefijo de $\mathcal{C}(b)$

Ejemplo

Para $\Sigma = \{a, b, c\}$ y $\mathcal{C}(a) = 0$, $\mathcal{C}(b) = 10$, $\mathcal{C}(c) = 11$, se tiene que $\mathcal{C}$ es libre de prefijos.

Corolario

Si $\mathcal{C}$ es una codificación libre de prefijos, entonces $\mathcal{C}^*$ es una función inyectiva.

Frecuencias relativas de los símbolos

String a codificar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\mathsf{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w

- Por ejemplo, si $w = aabaacaab$, entonces $\mathsf{fr}_w(a) = \frac{2}{3}$, $\mathsf{fr}_w(b) = \frac{2}{9}$ y $\mathsf{fr}_w(c) = \frac{1}{9}$

Para una Σ-codificación $\mathcal{C}$ definimos el largo promedio de la codificación de un símbolo de w (en bits) como:

$$\mathsf{lp}_w(\mathcal{C}) \quad = \quad \sum_{a \in \Sigma} \mathsf{fr}_w(a) \cdot |\mathcal{C}(a)|$$

Tenemos entonces que $|\mathcal{C}^*(w)| = \mathsf{lp}_w(\mathcal{C}) \cdot |w|$

- Por lo tanto queremos una Σ-codificación $\mathcal{C}$ que minimice $\mathsf{lp}_w(\mathcal{C})$

Frecuencias relativas de los símbolos y la función objetivo

Problema de optimización a resolver

Dado $w \in \Sigma^*$, encontrar una Σ-codificación $\mathcal{C}$ libre de prefijos que minimice el valor $\mathsf{lp}_w(\mathcal{C})$

La función objetivo del algoritmo codicioso es entonces $\mathsf{lp}_w(\cdot)$

- Queremos encontrar la Σ-codificación que minimice el valor de esta función
- Fue un problema abierto desde $\sim$ 1948
  - Resuelto en 1952 por David Huffman, estudiante del MIT

Frecuencias relativas de los símbolos y la función objetivo

La función $\mathsf{lp}_w(\cdot)$ se define a partir de la función $\mathsf{fr}_w(\cdot)$

- No se necesita más información sobre w. En particular, no se necesita saber cuál es el símbolo de w en una posición específica

Podemos entonces trabajar con funciones de frecuencias relativas en lugar de palabras

- La entrada del problema no va a ser w sino que fr_w

Frecuencias relativas de los símbolos y la función objetivo

Decimos que $f : \Sigma \to (0, 1)$ es una función de frecuencias relativas para Σ si se cumple que $\sum_{a \in \Sigma} f(a) = 1$

Dada una función f de frecuencias relativas para Σ y una Σ-codificación $\mathcal{C}$, el largo esperado de $\mathcal{C}$ para f se define como:

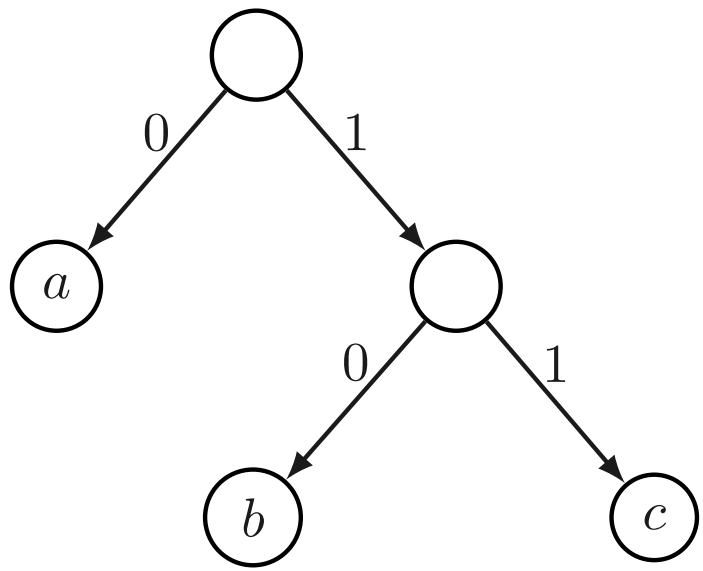
$$\mathsf{lp}_f(\mathcal{C}) \quad = \quad \sum_{a \in \Sigma} f(a) \cdot |\mathcal{C}(a)|$$

Una instancia del problema (entrada del algoritmo) es entonces una función f de frecuencias relativas para Σ, y la función objetivo a minimizar es $\mathsf{lp}_f(\cdot)$

Un ingrediente fundamental: codificaciones como árboles

Una manera simple de obtener una Σ-codificación es mediante árboles binarios, o *árboles de codificación*

Representamos la codificación $\mathcal{C}(a) = 0$, $\mathcal{C}(b) = 10$, $\mathcal{C}(c) = 11$ como un árbol binario:



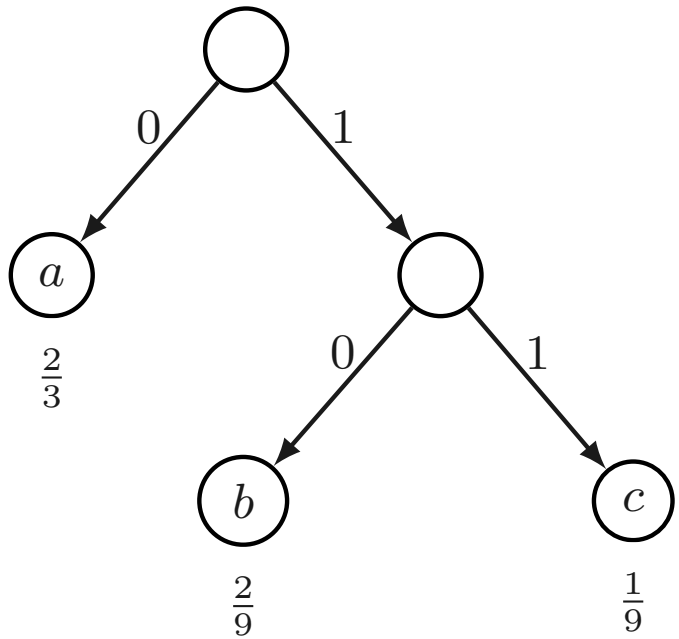
Un ingrediente fundamental: codificaciones como árboles

Si una Σ-codificación $\mathcal{C}$ es libre de prefijos, entonces el árbol que la representa satisface las siguientes propiedades.

- Cada hoja tiene como etiqueta un elemento de Σ, y estos son los únicos nodos con etiquetas
- Cada símbolo de Σ es usado exactamente una vez como etiqueta
- Cada arco tiene etiqueta 0 ó 1
- Si una hoja tiene etiqueta e y las etiquetas de los arcos del camino desde la raíz hasta esta hoja forman una palabra $w \in \{0,1\}^*$, entonces $\mathcal{C}(e) = w$

Codificaciones como árboles y las frecuencias relativas

Podemos agregar al árbol binario que representa una codificación $\mathcal{C}$ la información sobre las frecuencias relativas dadas por una función f:



Llamamos a este árbol abf$(\mathcal{C}, f)$

- ¡Estos árboles son muy útiles!

Alguna propiedades importantes

Ejercicios

Sea f una función de frecuencias relativas para Σ y $\mathcal{C}$ una Σ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_f(\cdot)$:

1. Sean u y v dos hojas en $\mathrm{abf}(\mathcal{C}, f)$ con etiquetas a y b, respectivamente. Demuestre que si el camino de la raíz a u es más corto que el camino de la raíz a v, entonces $f(a) \geqslant f(b)$

2. Demuestre que cada nodo interno en $\mathrm{abf}(\mathcal{C}, f)$ tiene dos hijos

3. Sean $a, b \in \Sigma$ tales que $a \neq b$, $f(a) \leqslant f(b)$ y $f(b) \leqslant f(e)$ para todo $e \in (\Sigma \setminus \{a, b\})$. Demuestre que existe una Σ-codificación $\mathcal{C}'$ libre de prefijos tal que $\mathrm{lp}_f(\mathcal{C}') = \mathrm{lp}_f(\mathcal{C})$ y las hojas con etiquetas a y b en $\mathrm{abf}(\mathcal{C}', f)$ son hermanas
   - Vale decir, existe $w \in \{0, 1\}^*$ tal que $\mathcal{C}'(a) = w0$ y $\mathcal{C}'(b) = w1$

Calculando el mínimo de $\mathrm{lp}_f(\cdot)$

Vamos a ver un algoritmo codicioso para calcular una Σ-codificación $\mathcal{C}$ libre de prefijos que minimiza $\mathrm{lp}_f(\cdot)$

- El algoritmo calcula la codificación de Huffman

El algoritmo tiene los ingredientes de un algoritmo codicioso.

- Función objetivo a minimizar: $\mathrm{lp}_f(\cdot)$
- Función de selección: elige los dos símbolos de Σ con menor frecuencia relativa, los coloca como hermanos en el árbol binario que representa la Σ-codificación óptima, y continua la construcción con el resto de los símbolos de Σ

Además, es necesario realizar una demostración para probar que el algoritmo es correcto.

El algoritmo de Huffman

En el siguiente algoritmo representamos a las funciones como conjuntos de pares ordenados, y suponemos que f es una función de frecuencias relativas que al menos tiene dos elementos en el dominio.

CalcularCodificaciónHuffman(f)

Sea Σ el dominio de la función f

if $\Sigma = \{a, b\}$ **then return** $\{(a, 0), (b, 1)\}$

else

Sean $a, b \in \Sigma$ tales que $a \neq b$, $f(a) \leqslant f(b)$ y
$f(b) \leqslant f(e)$ para todo $e \in (\Sigma \setminus \{a, b\})$

Sea c un símbolo que **no** aparece en Σ

$g := (f \setminus \{(a, f(a)), (b, f(b))\}) \cup \{(c, f(a) + f(b))\}$

$\mathcal{C}' :=$ **CalcularCodificaciónHuffman**(g)

$w := \mathcal{C}'(c)$

$\mathcal{C} := (\mathcal{C}' \setminus \{(c, w)\}) \cup \{(a, w0), (b, w1)\}$

return $\mathcal{C}$

La correctitud de **CalcularCodificaciónHuffman**

Teorema

*Si f es una función de frecuencias relativas, Σ es el dominio de f y $\mathcal{C} =$ **CalcularCodificaciónHuffman**(f), entonces $\mathcal{C}$ es una Σ-codificación libre de prefijos que minimiza la función $lp_f(\cdot)$*

Demostración: Vamos a realizar la demostración por inducción en $|\Sigma|$

Si $|\Sigma| = 2$, entonces la propiedad se cumple trivialmente

- ¿Por qué?

Suponga entonces que la propiedad se cumple para un valor $n \geqslant 2$, y suponga que $|\Sigma| = n + 1$

La demostración del teorema

Sean a, b, c, g, $\mathcal{C}'$ y $\mathcal{C}$ definidos como en el código de la llamada **CalcularCodificaciónHuffman**(f), y sea Γ el dominio de g

Como $|\Gamma| = n$ y $\mathcal{C}' =$ **CalcularCodificaciónHuffman**(g), por hipótesis de inducción tenemos que $\mathcal{C}'$ es una Γ-codificación libre de prefijos que minimiza la función $\mathsf{lp}_g(\cdot)$

Dada la definición de $\mathcal{C}$ es simple verificar las siguientes propiedades:

- $\mathcal{C}$ es una codificación libre de prefijos
- Para cada $e \in (\Sigma \setminus \{a, b\})$ se tiene que $\mathcal{C}(e) = \mathcal{C}'(e)$
- $|\mathcal{C}(a)| = |\mathcal{C}(b)| = |\mathcal{C}'(c)| + 1$

La demostración del teorema

Por contradicción suponga que $\mathcal{C}$ no minimiza el valor de la función $\mathsf{lp}_f(\cdot)$

- Vale decir, existe una Σ-codificación $\hat{\mathcal{C}}$ libre de prefijos tal que $\hat{\mathcal{C}}$ minimiza la función $\mathsf{lp}_f(\cdot)$ y por lo tanto $\mathsf{lp}_f(\hat{\mathcal{C}}) < \mathsf{lp}_f(\mathcal{C})$

Por la definición de a, b y los ejercicios anteriores podemos suponer que existe $w \in \{0,1\}^*$ tal que $\hat{\mathcal{C}}(a) = w0$ y $\hat{\mathcal{C}}(b) = w1$

- Nótese que $w \neq \varepsilon$ puesto que $|\Sigma| \geqslant 3$

A partir de $\hat{\mathcal{C}}$ defina la siguiente Γ-codificación $\hat{\mathcal{C}}'$:

$$\hat{\mathcal{C}}' \quad = \quad (\hat{\mathcal{C}} \setminus \{(a, \hat{\mathcal{C}}(a)), (b, \hat{\mathcal{C}}(b))\}) \cup \{(c, w)\}$$

Tenemos que $\hat{\mathcal{C}}'$ es una Γ-codificación libre de prefijos

- ¿Por qué?

La relación entre $\mathsf{lp}_g(\mathcal{C}')$ y $\mathsf{lp}_f(\mathcal{C})$

$$
\begin{aligned}
\mathsf{lp}_g(\mathcal{C}') &= \sum_{e\in\Gamma} g(e)\cdot|\mathcal{C}'(e)| \\
&= \left(\sum_{e\in(\Gamma\smallsetminus\{c\})} g(e)\cdot|\mathcal{C}'(e)|\right) + g(c)\cdot|\mathcal{C}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\mathcal{C}(e)|\right) + (f(a)+f(b))\cdot|\mathcal{C}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\mathcal{C}(e)|\right) + f(a)\cdot|\mathcal{C}'(c)| + f(b)\cdot|\mathcal{C}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\mathcal{C}(e)|\right) + f(a)\cdot(|\mathcal{C}(a)|-1) + f(b)\cdot(|\mathcal{C}(b)|-1) \\
&= \left(\sum_{e\in\Sigma} f(e)\cdot|\mathcal{C}(e)|\right) - (f(a)+f(b)) \\
&= \mathsf{lp}_f(\mathcal{C}) - (f(a)+f(b))
\end{aligned}
$$

La relación entre $\mathsf{lp}_g(\hat{\mathcal{C}}')$ y $\mathsf{lp}_f(\hat{\mathcal{C}})$

$$
\begin{aligned}
\mathsf{lp}_g(\hat{\mathcal{C}}') &= \sum_{e\in\Gamma} g(e)\cdot|\hat{\mathcal{C}}'(e)| \\
&= \left(\sum_{e\in(\Gamma\smallsetminus\{c\})} g(e)\cdot|\hat{\mathcal{C}}'(e)|\right) + g(c)\cdot|\hat{\mathcal{C}}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\hat{\mathcal{C}}(e)|\right) + (f(a)+f(b))\cdot|\hat{\mathcal{C}}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\hat{\mathcal{C}}(e)|\right) + f(a)\cdot|\hat{\mathcal{C}}'(c)| + f(b)\cdot|\hat{\mathcal{C}}'(c)| \\
&= \left(\sum_{e\in(\Sigma\smallsetminus\{a,b\})} f(e)\cdot|\hat{\mathcal{C}}(e)|\right) + f(a)\cdot(|\hat{\mathcal{C}}(a)|-1) + f(b)\cdot(|\hat{\mathcal{C}}(b)|-1) \\
&= \left(\sum_{e\in\Sigma} f(e)\cdot|\hat{\mathcal{C}}(e)|\right) - (f(a)+f(b)) \\
&= \mathsf{lp}_f(\hat{\mathcal{C}}) - (f(a)+f(b))
\end{aligned}
$$

Obteniendo una contradicción

Tenemos entonces que

$$\begin{aligned} \mathsf{lp}_g(\hat{\mathcal{C}}') &= \mathsf{lp}_f(\hat{\mathcal{C}}) - (f(a) + f(b)) \\ &< \mathsf{lp}_f(\mathcal{C}) - (f(a) + f(b)) \\ &= \mathsf{lp}_g(\mathcal{C}') \end{aligned}$$

Concluimos entonces que $\mathcal{C}'$ no minimiza la función $\mathsf{lp}_g(\cdot)$, lo cual contradice la hipótesis de inducción. □

Dos comentarios finales

La siguiente función calcula la codificación de Huffman teniendo como entrada una palabra w:

CalcularCodificaciónHuffman(w)
    if $w = \varepsilon$ **then return** $\varnothing$
    else
        $\Sigma :=$ conjunto de símbolos mencionados en w
        if $\Sigma = \{a\}$ **then return** $\{(a, 0)\}$
        else return CalcularCodificaciónHuffman(fr_w)

¿Cuál es la complejidad de **CalcularCodificaciónHuffman**(f)?

- ¿Qué estructura de datos debería ser utilizada para almacenar f?