Algoritmos de intersección y unión de conjuntos

Capítulo 6 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Algoritmos de intersección y unión de conjuntos en el modelo de comparación

1.1. Problema

Como se vio en capítulos anteriores, un conjunto es una colección de elementos donde no hay repetición. El uso de conjuntos es fundamental para un gran número de problemas. En particular, en este capítulo representaremos conjuntos como arreglos ordenados de números enteros; esto para posicionarlo dentro de un dominio de aplicación objetivo, que es la Recuperación de Información, como parte de la representación de una matriz dispersa muy grande, llamada *índice invertido*.

Resolveremos los problemas de intersección y unión de conjuntos. Demaine et al. (2000) demuestra que el costo y procedimiento de las intersecciones y uniones de conjuntos representados como arreglos

ordenados es básicamente el mismo, ya que requieren determinar la misma información. Claramente, recolectar los datos para la unión y la intersección requiere diferentes esfuerzos.

1.1.1. Costo del problema

En general, dados dos conjuntos $A[1..m] = \{a_1 < a_2 < \cdots < a_m\}$ y $B[1..n] = \{b_1 < b_2 < \cdots < b_n\}$, el costo de unión es

$$\log \binom{m+n}{m},$$

ver Hwang y Lin (1971).

De manera más detallada, supongamos que $A \cap B = \emptyset$; esto es, el conjunto de salida será de tamaño $m + n$. De manera similar al razonamiento que se utilizó para el problema de ordenamiento, el problema puede verse como todas las posibles instancias de órdenes o permutaciones de tamaño $n + m$; removiendo la necesidad de los órdenes parciales, esto es $\binom{n+m}{m}$ posibles instancias de tamaño $n + m$, generadas por dos conjuntos de tamaño n y m. Dado que estamos en un modelo basado en comparaciones y que el mejor algoritmo puede dividir el espacio de posibles órdenes entre 2, por lo tanto, dicho algoritmo necesitará

$$\log_2 \binom{n+m}{m}$$

comparaciones para resolver la unión de cualquier par de conjuntos de tamaño m y n.

Usando la aproximación de Stirling para coeficientes binomiales de MacKay (2003), el costo se convierte en:

$$\log \binom{m+n}{m} = n \log \frac{m+n}{n} + m \log \frac{n+m}{m}.$$

[1]

1.2. Algoritmos

Se puede observar que si $m \approx n$, entonces el costo se convierte en $O(m + n)$, esto es, lo más eficiente sería tomar el siguiente algoritmo:

```
function merge2!(C, A, B)                                    ①
    i = j = 1
    m, n = length(A), length(B)

    @inbounds while i <= m ||  j <= n                        ②
        a, b = A[i], B[j]
        if a == b
```

[1] Recuerde que $\log_2 x = \frac{\log_e x}{\log e}$.

```
            push!(C, a)
            i += 1
            j += 1
        elseif a < b
            push!(C, a)
            i += 1
        else
            push!(C, b)
            j += 1
        end
    end

    C
end
```

```
merge2! (generic function with 1 method)
```

1.2.1. Ejercicio

- Escriba y pruebe el algoritmo de intersección de merge para $n \approx m$.

1.3. Algoritmos para arreglos de tamaño muy diferente

Si $m \ll n$, el costo tenderá a $O(m \log n)$, por lo que se pueden realizar m búsquedas binarias *directas* para localizar la posición de inserción en B.

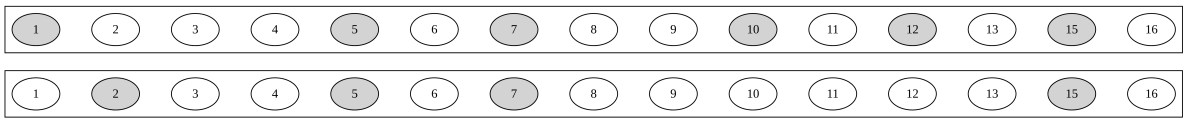

Figura 1: Dos listas alineadas donde los nodos sombreados son elementos de los conjuntos.

Se hace notar que A y B están ordenados, y por lo tanto, localizar $A[i]$ en $B[j]$ significa que $B[j-1] < A[i]$, por lo que intentar localizar $A[i+1]$ puede comenzar en $B[j+1]$. A continuación se muestra el código de un algoritmo de intersección usando algoritmos de búsqueda con memoria de la posición anterior.

```
function intsearch!(C, A, B, algosearch=doublingsearch)
    if length(B) < length(A)
        A, B = B, A
    end

    p = 1
```

```
    for (i, a) in enumerate(A)
        p = algosearch(B, a, p)
        p > length(B) && break
        if a == B[p]
            p += 1
            push!(C, a)
        end
    end

    C
end
```

Hwang y Lin (1971) propone otro algoritmo que funciona para casos similares:

1. Divide B en bloques de tamaño m, define un arreglo *virtual* $B'[1..n/m]$ donde $B'[i] = B[i \cdot m]$.
2. Se busca la posición de inserción p de cada $a \in A$ en B', costando $\log(n/m)$ para cada búsqueda.
3. Después se localiza dentro de B en el p-ésimo bloque, i.e., $B[(p-1)m + 1..p \cdot m]$, la posición de inserción del bloque, con un costo de $\log m$.

Entonces, se obtiene un costo de $O(m \log n/m + m \log m)$; esto es equivalente en el peor caso a búsquedas directas, i.e., cuando las posiciones de inserción de $a \in A$ se encuentran distribuidas de manera uniforme en B. Sin embargo, es posible mejorar si se descartan bloques en el paso 1. Esto es, si hay concentración de elementos de A en bloques de B. Para esto, es necesario un análisis de costo promedio, el cual se muestra en el artículo.

Incluso cuando hay concentración, podemos recordar la $(i-1)$-ésima posición de inserción para iniciar la i-ésima búsqueda, y sacar provecho de posiciones esperadas cercanas a la posición inicial de búsqueda, i.e., podemos utilizar algoritmos de búsqueda adaptables para mejorar el desempeño.

1.3.1. Algoritmo de Baeza Yates

Baeza-Yates (2004) propone un algoritmo eficiente para intersecciones de dos conjuntos. El algoritmo tiene una estrategia *dividir para vencer*:

1. Se toma la mediana M de A y se busca en B obteniendo su posición de inserción p.
2. El problema entonces se divide en 3 subproblemas:

$$C_< = \{A[1..M-1] \cap B[1..p-e]\} \tag{1}$$
$$C_= = \{A[M]\} \cap \{B[p]\} \tag{2}$$
$$C_> = \{A[M+1..m] \cap B[p+e..n]\} \tag{3}$$
$$\tag{4}$$

   donde $e = 1$ si $A[M] = B[p]$ y $e = 0$ cuando $A[M] \neq B[p]$.
3. La unión de estos tres conjuntos es la solución $C_< \cup C_= \cup C_>$.
4. El problema $C_=$ es trivial, y $C_<$ y $C_>$ se resuelven de forma recursiva, ajustando los rangos de trabajo.

A continuación se muestra el código en Julia, usando los algoritmos de búsqueda del Cap. 5.

```
# Adaptado de https://github.com/sadit/Intersections.jl

function baezayates!(output, A, B, findpos::Function=binarysearch)                    ①
    baezayates!(output, A, 1, length(A), B, 1, length(B), findpos)
end

function baezayates!(output, A, a_sp::Int, a_ep::Int, B, b_sp::Int, b_ep::Int, findpos::Functio
    (a_ep < a_sp || b_ep < b_sp) && return output
    imedian = ceil(Int, (a_ep + a_sp) / 2)
    median = A[imedian]
    ## our findpos returns n + 1 when median is larger than B[end]
    medpos = min(findpos(B, median, b_sp), b_ep)                                      ②

    matches = median == B[medpos]
    baezayates!(output, A, a_sp, imedian - 1, B, b_sp, medpos - matches, findpos)     ③
    matches && push!(output, median)                                                  ④
    baezayates!(output, A, imedian + 1, a_ep, B, medpos + matches, b_ep, findpos)     ⑤
    output
end
```

① Punto de entrada.
② Búsqueda de la posición de inserción de la mediana de A en B.
③ Recurrencia para el problema $C_<$.
④ Añadir al resultado el valor de la mediana si es que se encontró en B; es importante que este paso esté entre las recurrencias para que *output* sea un arreglo ordenado.
⑤ Recurrencia para el problema $C_>$.

El algoritmo de Baeza Yates es óptimo en el peor caso y es capaz de aprovechar casos donde $C_<$ o $C_>$ se convierten en triviales, lo cual da muy buenos casos en algunas distribuciones.

1.3.2. Ejercicios

1. Implemente la unión con el algoritmo de Baeza Yates.

1.4. Operaciones con tres o más conjuntos

Los algoritmos y costos hasta ahora revisados se cumplen para dos conjuntos; se mencionaron diferentes algoritmos, algunos de ellos especializados por características como las proporciones de los conjuntos de entrada.

En particular, es importante hacer notar que ni el problema ni las aplicaciones están limitadas a dos conjuntos y, por lo tanto, son importantes los algoritmos y estrategias para resolver $\bigcup_i A_i$ así como $\bigcap_i A_i$.

1.4.1. Algoritmo SvS

Dado que $C = A \cap B$, es un hecho que $|C| \leq \text{mín}\{|A|, |B|\}$. Recordando que hay maneras relativamente simples y eficientes de resolver la intersección cuando $m \ll n$, cuando tenemos más de dos conjuntos podemos aplicar la estrategia *Small vs Small (SvS)*, que consiste en intersectar los k conjuntos por pares intersectando el par de arreglos más pequeños cada vez.

```
# Adaptado de https://github.com/sadit/Intersections.jl

function svs(L::Vector{T}, in2::Function=baezayates!) where T
    prev, curr = eltype(T)[], eltype(T)[]
    sort!(L, by=length, rev=true)
    curr = pop!(L)

    while length(L) > 0
        empty!(prev)
        isize = in2(prev, curr, pop!(L))
        isize == 0 && return prev
        prev, curr = curr, prev
    end

    curr
end
```

1.4.2. Algoritmo de Barbay y Kenyon

Existe otra familia de algoritmos, basados en búsquedas adaptativas que pueden llegar a mejorar el desempeño bajo cierto tipo de entradas. Demaine et al. (2001), Barbay et al. (2006) y Barbay et al. (2010) muestran algoritmos de intersección basados en búsquedas adaptativas para aprovechar instancias simples. Estos estudios se basan en contribuciones teóricas de los mismos autores: Demaine et al. (2000), Demaine et al. (2001), Barbay y Kenyon (2002) y Baeza-Yates (2004).

El algoritmo de Barbay et al. (2006) trabaja sobre los k conjuntos de entrada, representados como arreglos ordenados de números enteros. Es un algoritmo simple pero poderoso: hace uso de búsquedas adaptativas con memoria para guardar las posiciones donde se avanza, de tal forma que no se recalculen posiciones. Las diferentes estrategias para revisar los conjuntos pueden dar diferentes desempeños, como se valida en Barbay et al. (2010), donde además de hacer una gran variedad de experimentos sobre diferentes algoritmos de búsqueda, se introducen variantes en el orden de acceso de cada conjunto.

A continuación se muestra el código del algoritmo base:

```
# Adaptado de https://github.com/sadit/Intersections.jl

function bk!(output, L::AbstractVector, findpos::Function=doublingsearch)
    P = ones(Int, length(L))
    bk!(output, L, P, findpos)
```

```
end

function bk!(output, L, P, findpos::Function=doublingsearch)                    ①
    n = length(L)                                                               ②
    el = L[1][1]                                                                ③
    c = 0                                                                       ④

    @inbounds while true
        for i in eachindex(P)
            P[i] = findpos(L[i], el, P[i])                                      ⑤
            P[i] > length(L[i]) && return output
            pval = L[i][P[i]]
            if pval == el
                c += 1
                if c == n                                                       ⑥
                    push!(output, el)
                    c = 0                                                       ⑦
                    P[i] += 1
                    P[i] > length(L[i]) && return output
                    el = L[i][P[i]]
                end
            else
                c = 0
                el = pval
            end
        end
    end

    output
end
```

① El algoritmo de Barbay & Kenyon recibe: i) `output` el conjunto de salida. ii) `L` la lista de conjuntos (representados como arreglos ordenados). iii) `P` arreglo de posiciones *actuales* para cada arreglo. iv) `findpos` función de búsqueda.
② Número de conjuntos en `L`.
③ `el` es el elemento siendo buscado en todos los arreglos.
④ `c` número de listas que contienen `el`.
⑤ Buscando la posición de inserción de `el` en `L[i]`, comenzando en `P[i]`.
⑥ Esta igualdad implica que hay intersección.
⑦ Reiniciando `el` y `c` y actualizando `P[i]`.

De manera particular, Barbay et al. (2010) presentan un estudio experimental sobre los algoritmos presentados en el área durante la década de 2000 a 2010, dichos algoritmos se parametrizaron de maneras que nos permiten aprender diferentes características de cada uno de ellos, dependiendo de los algoritmos de búsqueda que usan, la arquitectura computacional donde se evalúa, y el número de conjuntos siendo procesados.

1.5. Recursos audio-visuales de la unidad

Parte 1: Algoritmos de intersección (y unión) de listas ordenadas

Parte 2: Algoritmos de intersección y algunas aplicaciones

1.6. Actividades

1. Lea el archivo de datos provisto, el cual se encuentra en formato JSON, y contiene múltiples listas de datos:
   - Conjunto A que contiene pares de listas.
   - Conjunto B que contiene tripletas de listas.
   - Conjunto C que contiene tuplas de 4 listas.
2. Implementación de algoritmos:
   - Implementa los algoritmos de Melding (ME), Baeza-Yates (BY) y de Barbay & Kenyon (BK).
   - Implementa BY parametrizando con algoritmos de búsqueda: bisección y búsqueda no acotada B_1 y B_2 (ver Capítulo 5).
3. Experimentación:
   - Calcula las intersecciones para los datos de cada grupo de listas (A, B, y C).
   - Acumula el tiempo en segundos y el número de comparaciones para cada operación, es posible que necesites repetir varias veces el procedimiento para medir tiempos de manera fiable.
   - Crea gráficos boxplot para cada conjunto de listas con los siguientes aspectos, incluye en cada gráfica todos los algoritmos para que sean fáciles de comparar:
     - Tiempo en segundos de los tres experimentos, por algoritmo.
     - Número de comparaciones de los tres experimentos, por algoritmo.
     - Longitudes de las intersecciones resultantes para A, B, C, este también es un gráfico de control, úsalo para detectar posibles errores.
4. Análisis:
   - Analiza los resultados obtenidos en los gráficos boxplot. Examina las diferencias en tiempo, número de comparaciones y longitud de las intersecciones entre los distintos algoritmos y conjuntos de listas.
   - Discute cualquier tendencia, anomalía o patrón interesante observado. Reflexiona sobre cómo los diferentes algoritmos afectan el rendimiento y la eficiencia en las operaciones de intersección.

1.6.1. Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos:

- Título del reporte, su nombre.
- Introducción.
- Desarrollo de las actividades.
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

Baeza-Yates, Ricardo. 2004. "A fast set intersection algorithm for sorted sequences". *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings 15*, 400–408.

Barbay, Jérémy, y Claire Kenyon. 2002. "Adaptive intersection and t-threshold problems". *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA), SODA '02, 390–99.

Barbay, Jérémy, Alejandro López-Ortiz, y Tyler Lu. 2006. "Faster adaptive set intersections for text searching". *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings 5*, 146–57.

Barbay, Jérémy, Alejandro López-Ortiz, Tyler Lu, y Alejandro Salinger. 2010. "An experimental investigation of set intersection algorithms for text searching". *Journal of Experimental Algorithmics (JEA)* 14: 3–7.

Demaine, Erik D, Alejandro López-Ortiz, y J Ian Munro. 2001. "Experiments on adaptive set intersections for text retrieval systems". *Algorithm Engineering and Experimentation: Third International Workshop, ALENEX 2001 Washington, DC, USA, January 5–6, 2001 Revised Papers 3*, 91–104.

Demaine, Erik D, Alejandro López-Ortiz, y J Ian Munro. 2000. "Adaptive set intersections, unions, and differences". *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, 743–52.

Hwang, Frank K., y Shen Lin. 1971. "Optimal merging of 2 elements with n elements". *Acta Informatica* 1 (2): 145–58.

MacKay, David JC. 2003. *Information theory, inference and learning algorithms*. Cambridge university press.