Balanced Shortest Supersequence (BSS)

Hansel Blanco, Elena Rodríguez

3 de abril de 2023

1. Descripción del problema

Dadas dos cadenas de paréntesis abiertos y cerrados A y B, no necesariamente balanceadas, se quiere encontrar una cadena C de tamaño mínimo tal que contenga a las cadenas A y B como subsecuencias (no necesariamente continuas) y además este balanceada.

Se garantiza que las cadenas A y B no tendrán más de 300 paréntesis abiertos ni de 300 paréntesis cerrados.

Una cadena de paréntesis abiertos y cerrados está balanceada si cumple lo siguiente:

1. La cadena '()' está balanceada.
2. Si a una cadena balanceada se le añade un '(' al principio y un ')' al final el resultado sigue siendo una cadena balanceada.
3. La concatenación de dos cadenas balanceadas es una cadena balanceada.

Llamaremos a la cadena solución C: supersecuencia balanceada de A y B de tamaño mínimo.

2. Solución óptima que se propone

La solución óptima que se propone es una solución con programación dinámica, aprovechando la subestructura óptima de la supersecuencia solución.

Se utiliza como estructura una tabla de 3 dimensiones (dp). Una dimensión para representar cada cadena de entrada y otra dimensión que representa para cada par de prefijos de A y B todos los posibles factores de balance de las supersecuencias de dichos prefijos. En cada posición de la tabla se termina poniendo la longitud de la menor supersecuencia de los prefijos correspondientes de A y B que tiene factor de balance (FB) k, esto se logra a partir de una serie de relajaciones que se describen más adelante.

Inicialmente todas las posiciones de la tabla se inicializan en infinito (10^{10} que sabemos que las supersecuencias nunca tendrán ese factor de balance) para garantizar que sea más grande que todas las longitudes que pueden ir apareciendo en la formación de las supersecuencias.

Como caso base, en la posición [0][0][0] de la tabla se inserta el valor 0, pues la menor supersecuencia de los prefijos de tamaño 0 de A y B, que tiene $FB = 0$ es la cadena vacía cuya longitud es igual a cero. Y no tiene sentido para nuestro problema construir supersecuencias de prefijos de tamaño cero y $FB > 0$, porque serían caracteres innecesarios y se está buscando longitud mínima.

Para llenar la tabla se itera sobre los índices de la estructura de forma tal que, para cada par de prefijos de A y B, se exploran todos los factores de balance posibles (toda la tercera dimensión).

Se define como relajación de una casilla de la tabla (dp[i][j][k]), a la actualización del valor de dicha casilla al encontrarse una supersecuencia de los prefijos que ella representa con el mismo FB y que es menor en longitud.

Durante la iteración por los índices de la estructura, a partir de la posición indexada en i,j,k, se intentan relajar las casillas a las que se puede llegar añadiendo el siguiente caracter de A, de B o de ambos si son iguales. Esto es: $dp[i+1][j][k_1]$, $dp[i][j+1][k_2]$ y $dp[i+1][j+1][k_3]$, donde k_1 es el nuevo FB tras añadir el caracter $i+1$ de A, k_2 es el nuevo FB tras añadir el carácter $j+1$ de B y k_3 es nuevo FB tras añadir el caracter $i+1$ de A que es igual al $j+1$ de B.

Si el caracter que se quiere añadir es '(' entonces el FB de la supersecuencia aumenta en 1, y por tanto el índice de la tercera dimensión se computa como k+1.
Si el caracter que se quiere añadir es ')' entonces el FB de la supersecuencia disminuye en 1 si era mayor que cero, y por tanto el índice de la tercera dimensión se computa como k-1.
En estos dos casos se está añadiendo un caracter a la supersecuencia y por tanto el valor que se intenta poner es dp[i][j][k] + 1.
Si se intenta añadir un paréntesis cerrado cuando FB=0 entonces el FB se queda igual pues no hemos añadido un nuevo '(', ni cerrado un '(' que no estaba cerrado. En este caso el valor que se intenta poner en la casilla es dp[i][j][k] + 2, pues sabemos que tenemos que añadir obligatoriamente 2 caracteres a la supersecuencia.

2.1. Subestructura óptima

Sobre la estructura de la cadena solución podemos establecer lo siguiente:

Sean $A = (a_1, a_2, ..., a_n)$ y $B = (b_1, b_2, ..., b_m)$ las secuencias de entrada del problema, y sea $C = (c_1, c_2, ..., c_k)$ la supersecuencia de A y B, que no contiene paréntesis cerrados sin balancear, tal que balanceada es la supersecuencia de menor tamaño de A y B balanceada. Sea P un prefijo de C, P es la supersecuencia de un prefijo de A, sea A_i y un prefijo de B, sea B_j (donde i es la posición del último carácter de A que está en P, y j es lo mismo para B) tal que su longitud más la cantidad de paréntesis abiertos sin cerrar de P en C es lo menor posible.

Demostración:

Supóngase que no es así.

Definamos $C = P + C_1$. Como C es supersecuencia de A y B y P es supersecuencia de A_i y B_j entonces C_1 es supersecuencia de A desde la posicion i+1 hasta el final de A y de B desde j+1 hasta el final de B. Nótese que si esto no fuera así habría algún caracter de A desde la posicion i+1 hasta el final de A o de B desde j+1 hasta el final de B, que no estaría en C_1 y por definición de P no estaría tampoco en P, por tanto no estaría en C, lo que contradice el hecho de que C sea supersecuencia de A y B.

Sea P' una supersecuencia de A_i y B_j, que no contiene paréntesis cerrados sin balancear, tal que su longitud más la cantidad de paréntesis abiertos sin cerrar de P' en $P' + C_1$ es menor que los de P en C.

Sea $C' = P' + C_1$, C' es una supersecuencia de A y B pues, P' es supersecuencia de A_i y B_j y C_1 es supersecuencia de A desde la posicion i+1 hasta el final de A y de B desde j+1 hasta el final de B.

Definamos para la notación: $|FB_x|_y$ como la cantidad de paréntesis abiertos sin cerrar que tiene el prefijo x de la cadena y en la cadena y. Si x=y entonces se trata del factor de balance de la cadena x.

Sabemos que:
$|C'| + |FB_{C'}|_{C'} = |C| - |P| + |P'| + |FB_C|_C - |FB_P|_C + |FB_{P'}|_{C'}$
$= |C| + |FB_C|_C - ((|P| + |FB_P|_C) - (|P'| + |FB_{P'}|_{C'}))$

Como P' es una supersecuencia de A_i y B_j tal que su longitud más la cantidad de paréntesis abiertos sin cerrar de P' en $P'+C_1$ es menor que los de P en C, entonces: $((|P|+|FB_P|_C)-(|P'|+|FB_{P'}|_{C'})) \geq 1$, y por tanto:
$|C'| + |FB_{C'}|_{C'} = |C| - |P| + |P'| + |FB_C|_C - |FB_P|_C + |FB_{P'}|_{C'}$
$= |C| + |FB_C|_C - ((|P| + |FB_P|_C) - (|P'| + |FB_{P'}|_{C'})) \leq |C| + |FB_C|_C - 1 < |C| + |FB_C|_C$

Luego hemos encontrado una supersecuencia de A y B tal que su longitud más lo que le falta para estar balanceada es menor que la longitud de C más lo que le falta a C para estar balanceada. Es decir, C' es una supersecuencia de A y B que balanceada tiene menor longitud que C. Esto contradice el hecho de que C sea la menor, y por tanto lo supuesto es falso.

2.2. Correctitud

Se demostrará por inducción en el número de entradas al tercer ciclo del algoritmo que en la n-ésima entrada al tercer ciclo en la posición i_n, j_n y b_n está la longitud de la supersecuencia de menor ta-

maño de A_{i_n} y de B_{j_n} con factor de balance b_n y que este valor no se modifica en iteraciones posteriores.

Caso base: $n = 1$ En la primera entrada al tercer ciclo, los índices son todos iguales a cero. dp[0][0][0]= 0 en este momento, pues se asigno justo antes de entrar por primera vez al ciclo. Precisamente la longitud de la menor supersecuencia de A_0 y B_0 con FB=0, es la cadena vacía, cuya longitud es 0. Este valor nunca se modifica porque en el algoritmo nunca se actualiza una combinación de índices anterior o igual a la del momento, y todas son mayores o iguales que [0][0][0]

Hipótesis de inducción: Supongamos que para todo $k < n$, la tabla en los índices correspondientes a la iteración k (i_k, j_k, b_k) tiene la longitud de la supersecuencia de menor tamaño de A_{i_k} y de B_{j_k} con factor de balance b_k y que este valor no se modifica en lo sucesivo.

Paso inductivo: Sea la n-ésima entrada al tercer ciclo y sean i,j,b los índices correspondientes a dicha iteración.

$A[i][j][b]$, solo depende de:

$A[i-1][j][b], A[i-1][j][b-1], A[i-1][j][b+1]$

$A[i][j-1][b], A[i][j-1][b-1], A[i][j-1][b+1]$

$A[i-1][j-1][b], A[i-1][j-1][b-1], A[i-1][j-1][b+1]$

pues en el algoritmo la posición i,j,b solo puede ser alcanzada por una de estas posiciones lo cual es correcto pues al ser $A[i][j][b]$ la longitud de una supersecuencia de A_i y B_j con FB= b, el último caracter de dicha supersecuencia solo puede ponerse después de haber puesto los $i-1$ caracteres de A y los $j-1$ caracteres de B, y no puede haberse puesto un caracter de A posterior al de la posición i, ni uno de B posterior al de la posición j.

Como todas esas combinaciones de índices corresponden a iteraciones anteriores a la n-ésima por la forma en que se generan las combinaciones de índices en los tres ciclos anidados, entonces se puede aplicar la hipótesis de inducción en ellas. Además, se sabe que en cada una de esas iteraciones se hizo una relajación si fue posible en la casilla $A[i][j][b]$.

Supóngase, sin perder generalidad, que la última relajación se realizó desde la casilla $A[i-1][j][b]$, esto significa que la forma más corta de llegar hasta la casilla $A[i][j][b]$ es construir la supersecuencia hasta la casilla $A[i-1][j][b]$ y luego movernos de ella hacia la casilla objetivo. Como el valor de $A[i-1][j][b]$ es por hipótesis de inducción la longitud de la menor supersecuencia de A_{i-1} y B_j con FB= b, entonces en este momento de la ejecución del algoritmo $A[i][j][b]$ tiene como valor la longitud de la menor supersecuencia de A_i y B_j con factor de balance b.

Luego al terminar el algoritmo en las posiciones n, m, b donde n es la longitud de la cadena A, m es la longitud de la cadena B y b es cualquiera de los FB posibles se tiene la longitud de la menor supersecuencia de A y B con factor de balance b.

La cadena solución es la supersecuencia de A y B tal que balanceada es la menor en longitud. Sabemos que lo que le falta en longitud a las longitudes de supersecuencias de A y B construidas para estar balanceadas es su FB correspondiente. Por tanto la solución es la supersecuencia de A y B tal que su longitud sumado al factor de balance sea lo menor posible.

Luego, se construye la cadena solución creando un camino hacia la raíz del árbol de recorrido que se ha ido construyendo en cada relajación a casillas adyacentes. Por tanto, se inicia desde la casilla con el menor factor de balance de la lista de FB que corresponde a las cadenas A y B totalmente consumidas. A partir de este, se compara con los valores de las 9 casillas antes expuestas desde las que se relajó potencialmente esta casilla, al coincidir este valor reducido con su factor de balance en el paso anterior, se toma como siguiente elemento de la cadena solución (anterior si se quiere observar de

forma ordenada), y así sucesivamente hasta llegar a la posición [0, 0, 0]. Luego, esta cadena invertida con su correspondiente balance, es la supersecuencia solución.

2.3. Complejidad temporal

El costo temporal es la construcción de la tabla tridimensional de la propuesta, la primera dimensión es la cantidad de caracteres de la secuencia A sumado a 1 ($|A| + 1$), la segunda es la cantidad de caracteres de la secuencia B sumado a 1 ($|B| + 1$), y la tercera dimensión es de tamaño $|A| + |B| + 1$.

Luego, iterar esta matriz para asignar los valores en las posiciones necesarias $\in O(n * m * (n + m))$ donde $n = |A|$ y $m = |B|$. Además, se sabe que $n + m \leq 2max(n, m)$, y por tanto $n * m * (n + m) \leq n * m * 2max(n, m)$, por lo que $O(n * m * (n + m)) \in O(n * m * max(n, m))$. Por tanto, la complejidad temporal $T(n, m) \in O(n * m * max(n, m))$

3. Generador de casos de pruebas

Se ha implementado un generador de casos de prueba utilizando la biblioteca `random` de `Python`, como se muestra a continuación:

Listing 1: Random Generator de `tester.py`

```
def random_generator() -> Tuple[str, str]:

    alphabet = '()'
    a = b = ''

    size_a = rd.randint(1, 600)
    open_count = closed_count = 0
    for _ in range(size_a):
        number = rd.randint(0, 1)
        open_count += 1 if number == 0 else open_count
        closed_count += 1 if number == 1 else closed_count
        if open_count == 300 or closed_count == 300:
            break
        a += alphabet[number]

    size_b = rd.randint(1, 600)
    open_count = closed_count = 0
    for _ in range(size_b):
        number = rd.randint(0, 1)
        open_count += 1 if number == 0 else open_count
        closed_count += 1 if number == 1 else closed_count
        if open_count == 300 or closed_count == 300:
            break
        b += alphabet[number]

    return a, b
```

Se genera un tamaño aleatorio para cada cadena (hasta 600 caracteres, con límite de 300 para cada tipo de caracter). Luego son generados esta cantidad de caracteres del alfabeto `alphabet` con probabilidad uniforme.

4. Probador de casos

Cuando se ejecuta el archivo `tester.py` se muestra por cada solución propuesta el tiempo que tarda cada algoritmo en su ejecución, el tamaño de la secuencia solución y dicha secuencia. En consola la cadena A se muestra de color azul y la B de rojo. En el caso del algoritmo de fuerza bruta y el primer algoritmo recursivo (propuesta que cuenta paréntesis abiertos y cerrados) se muestran en azul los caracteres de A, en rojo los de B y en morado cuando el caracter pertenece a ambas cadenas, útil en el proceso de identificar patrones para los siguientes algoritmos propuestos. A continuación se muestra un fragmento de código que muestra lo antes expuesto:

Listing 2: Fragmento de código de `tester.py`

```
# brute force
start_bf = time.time()
result_bf = brute_force(a, b)
end_bf = time.time() - start_bf
print(Style.RESET_ALL, end = '')
print(len(result_bf[0]))
pretty_printing(result_bf[0], result_bf[1])
print(Style.RESET_ALL, end = '')
print('Time: ' + str(end_bf))


# upgraded brute force oc
start_ubf = time.time()
result_upgrade_bf = find_min_chain_oc(a, b)
end_ubf = time.time() - start_ubf
print(Style.RESET_ALL, end = '')
print(len(result_upgrade_bf[0]))
pretty_printing(result_upgrade_bf[0], result_upgrade_bf[1])
print(Style.RESET_ALL, end = '')
print('Time: ' + str(end_ubf))

# upgraded brute force bf
start_ubf = time.time()
result_upgrade_bf = find_min_chain_balance_f(a, b)
end_ubf = time.time() - start_ubf
print(Style.RESET_ALL, end = '')
print(len(result_upgrade_bf))
print(result_upgrade_bf)
print(Style.RESET_ALL, end = '')
print('Time: ' + str(end_ubf))


# dynamic programming
start_ubf = time.time()
chain = dp_bjk(a, b)
end_ubf = time.time() - start_ubf
print(len(chain))
print(Style.RESET_ALL, end = '')
print(chain)
print('Time: ' + str(end_ubf))
```

Para verificar la correctitud de las soluciones se propone una comparación del tamaño de estas, al ser válidas varias soluciones de la misma longitud. Se aconseja modificar en el generador de cadenas el tamaño límite de estas para observar la correctitud de los casos de prueba en todas las propuestas de solución.

5. Otras soluciones que se proponen

5.1. Solución 1

Esta propuesta consiste en generar todas las variaciones con repetición pasando por todos los tamaños de las posibles cadenas de forma creciente, hasta encontrar una cadena que esté balanceada y contenga de la forma definida en el ejercicio a A y B.

5.1.1. Correctitud

La propuesta explora todo el espacio de las cadenas de cualquier tamaño, por tanto, explora todo el espacio de soluciones, garantizando pasar por un tamaño que es suficiente para resolver el problema, y devuelve exactamente una cadena (la primera que se encuentra) del conjunto de las cadenas que son solución. Como se menciona, el espacio de búsqueda es explorado de forma creciente en los tamaños de las cadenas, por lo que en el primer tamaño que se encuentre una cadena que cumpla las restricciones del problema, esta cadena es una de las soluciones óptimas (de menor longitud) y por tanto, solución al problema.

5.1.2. Complejidad temporal

El costo temporal es el costo de generar todas las variaciones con repetición de todos los tamaños hasta un k de un conjunto de 2 elementos, donde k es el tamaño mínimo necesario para que la solución contenga ambas cadenas A y B y además, esté balanceado. Por tanto, el algoritmo $\in O(2^k * k)$, ya que para cada tamaño k existen 2^k variaciones con repetición que se pueden formar utilizando el conjunto de 2 elementos, y cada variación tiene una longitud de k. Entonces, se deben generar 2^k variaciones para cada tamaño de k desde 1 hasta k, y en cada iteración se deben hacer k asignaciones de elementos, lo que nos da una complejidad temporal total de $O(2^k * k)$.

5.2. Solución 2

Durante la construcción de la supersecuencia de A y B tal que balanceada es la menor posible se ha seguido la siguiente estrategia:

Construir una supersecuencia de A y B, sin añadir ningún caracter extra, y saber cuántos paréntesis abiertos y cerrados necesita que se le añadan para estar balanceada.

Luego de todas las supersecuencias balanceadas posibles, la solución será la de menor tamaño.

5.2.1. Correctitud

En cada momento de la construcción de la supersecuencia, tendremos dos punteros, i, que indica la posición del primer caracter de la cadena A que no se ha añadido a la supersecuencia y j que es lo análogo para la cadena B.

En cada momento se tienen a lo sumo 3 posibilidades:

1. Si i es menor que la longitud de A y j es menos que la longitud de B y A[i]= B[j], entonces se puede añadir a la cadena C el caracter A[i] y aumentar en 1 ambos punteros

2. Si i es menor que la longitud de A entonces se puede añadir a la cadena C el caracter A[i] y aumentar en 1 el puntero i.

3. Si j es menor que la longitud de B entonces se puede añadir a la cadena C el caracter B[j] y aumentar en 1 el puntero j.

Como se explora todo el espacio de búsqueda, entoces en las cadenas candidatas estará la supersecuencia de A y B que balanceada es la más corta posible. Esta es la que encuentra el algoritmo, al quedarse con la aquella que tiene menor longitud más necesidad de paréntesis abiertos y cerrados.

En cada llamado recursivo se llevan dos parámetros para la cadena en construcción, la cantidad de paréntesis abiertos y la cantidad de paréntesis cerrados necesarios para balancear la cadena que se tiene hasta el momento. Estos parámetros se actualizan de la siguiente forma:

1. Inicialmente la cadena es vacía y se considera balanceada, por lo que ambos parámetros son iguales a cero.

2. Cuando se añade un paréntesis abierto, entonces la cantidad de cerrados necesarios aumenta en 1 porque se necesita un ')' para balancear a dicho paréntesis abierto.

3. Cuando se añade un paréntesis cerrado, si la cantidad de cerrados necesarios es mayor que cero, la cantidad de cerrados necesarios disminuye en uno porque con este ')' se está cerrando un paréntesis abierto anterior y si la cantidad de cerrados necesarios es igual a cero, entonces la cantidad de abiertos necesarios aumenta en 1, porque se necesita un '(' para balancear a dicho ')'

De esta manera, al terminar de construir cada supersecuencia de A y B, los parámetros cantidad de paréntesis abiertos y la cantidad de paréntesis cerrados necesarios, tienen la cantidad de paréntesis abiertos y cerrados necesarios para balancear La conssecuenciaistencia de estos la superparámetros se justifica con la propia definición de cadena de paréntesis balanceada.

Se puede demostrar que si en un momento dado de la construcción de la cadena se puede tomar la decisión 1, el resultado nunca es peor que si se toman las soluciones 2 o 3, por lo que basta explorar las soluciones que se derivan de esa decisión. Con esto se reduce en algunos casos el espacio de búsqueda.

Demostración:

Sea C_1 una cadena candidata a ser solución tal que durante su construcción en un momento en el que era posible tomar la posibilidad 1 anteriormente descrita, se tomó otra de las posibilidades. Asumamos sin perder generalidad que se tomó la posibilidad 2.

Sean i y j los punteros de las cadenas A y B en ese momento. Se sabe que C_1 es de la forma:

$$C_1 = C_1' + a_i + ... + a_{i+p} + b_j + C_1''$$

donde:

1. p ≥ 0 es la cantidad de caracteres de A que hay entre a_i y b_j en la cadena C_1
2. C_1' es una supersecuencia del prefijo de la cadena A hasta la posición i-1 y del prefijo de B hasta la posición j-1, pues los caracteres se van añadiendo a la cadena en orden creciente se su aparición en las secuencias A y B
3. $a_i = b_j$
4. C_1'' es una supersecuencia del sufijo de la cadena A desde la posición i+p+1 y del prefijo de B desde la posición j+1, pues los caracteres se van añadiendo a la cadena en orden creciente se su aparición en las secuencias A y B y se añaden todos.

Sean on_{C1} y cn_{C1} la cantidad de paréntesis abiertos y cerrados respectivamente que necesita la cadena C_1 para balancearse:

Construyamos la cadena

$$C = C_1' + a_i + ... + a_{i+p} + C_1''$$

que solo se diferencia de C_1 en que cuando los punteros eran i y j se tomó la decisión 1, nótese que es posible hacer esto porque $a_i = b_j$ y que en cada momento posterior pueden seguirse tomando las decisiones que se tomaron en la construcción de C_1 (pueden ponerse los caracteres hasta a_{i+p} porque el último caracter de A que se puso fue i-ésimo y puede ponerse C_1'' porque en ese momento se han puesto exactamente los mismos caracteres de A y B que en C)

Demostremos que C es una supersecuencia de A y B que al balancearse tiene menor o igual cantidad de caracteres que C_1.

Por la forma en que está construida se sabe que C es una supersecuencia de A y B.

Además $|C| = |C_1| - 1$ y la cantidad de caracteres que C necesita para balancearse es a lo sumo 1 más de lo que necesita C_1 pues el hecho de que b_j no aparezca como caracter independiente de a_j puede provocar en el caso peor que quede desbalanceado un paréntesis que lo balanceaba b_j (un caracter no balancea a más de un caracter en una cadena, y por tanto eliminar un caracter no desbalancea a más de un caracter de la cadena).

Luego la cantidad de caracteres de la supersecuencia de A y B, C, al balancearse no es mayor que la de la supersecuencia C_1, por tanto, siempre puede tomarse la opción 1 y no se empeora el resultado.

Luego, en la implementación del algoritmo recursivo se ordenan los subproblemas de la siguiente forma:

1. Si i es menor que la longitud de A y j es menos que la longitud de B y A[i]= B[j], entonces se añade a la cadena C el caracter A[i] y aumentar en 1 ambos punteros

   Si esto no se cumple, entonces:

2. Si i es menor que la longitud de A entonces se puede añadir a la cadena C el caracter A[i] y aumentar en 1 el puntero i.
3. Si j es menor que la longitud de B entonces se puede añadir a la cadena C el caracter B[j] y aumentar en 1 el puntero j.

5.2.2. Complejidad temporal

La fórmula recursiva del algoritmo es:

$$T(n, m) = T(n-1, m) + T(n, m-1) + c$$

El valor c representa el costo de la comparación de caracteres y actualizaciones de variables en cada llamado recursivo, que es una cantidad constante de trabajo que involucra simples operaciones aritméticas, etc.

Luego, por el método recursivo del árbol se puede llegar a que en el nivel 1, hay dos llamadas recursivas, una para $T(n-1,m)$ y otra para $T(n,m-1)$. Cada una de estas llamadas recursivas genera un subárbol en el que los valores de n o m se reducen en 1. Por lo tanto, se forman dos subárboles en el siguiente nivel. En general, para el nivel i, habrá 2^i nodos, cada uno de los cuales exige una operación c. Por lo tanto, el tiempo total de ejecución $T(n,m) = c * 2^0 + c * 2^1 + ... + c * 2^{n+m}$.

Por tanto, $T(n,m) \in O(2^{n+m})$, lo que significa que su tiempo de ejecución crece exponencialmente con el tamaño de entrada.

5.3. Solución 3

Si se observa con cuidado la solución anterior se puede notar que:

1. El parámetro cantidad de paréntesis abiertos necesarios en una supersecuencia nunca disminuye. Estos serán caracteres que siempre habrá que añadir a una supersecuencia para balancearla.

2. El concepto de paréntesis cerrados necesarios es equivalente al concepto de factor de balance. Es la cantidad de paréntesis abiertos sin cerrar que hay en una cadena.

A partir de estas observaciones y como alternativa a la solución anterior se propone:

1. Si durante la construcción de una supersecuencia de las cadenas A y B, se va a añadir un paréntesis cerrados sin que haya un paréntesis abierto sin cerrar antes en la supersecuencia, entonces se añade el paréntesis cerrado y uno abierto delante para balancearlo. Es decir se añade '()'

2. Se sustituye el parámetro paréntesis cerrados necesarios por factor de balance

5.3.1. Correctitud

Demostremos que para toda supersecuencia de paréntesis de A y B balanceada, si existe un paréntesis cerrado que estaba en la supersecuencia antes de añadir caracteres para balancear y que no estaba balanceado, entonces hay una supersecuencia de A y B balanceada que tiene al paréntesis abierto que balancea a dicho paréntesis cerrado justo antes de él y que tiene la misma longitud que la primera supersecuencia balanceada mencionada.

Sea C una supersecuencia balanceada de A y B, sea S la supersecuencia C antes de añadir caracteres extra para balancear, sea c_i un caracter ')' de C que no estaba balanceado en S. Sea c_j el caracter '(' de C que es el '(' anterior a c_i y más cercano a él, que sabemos que existe porque C está balanceada, podemos escribir C como:

$$C = C_1 + c_j + C_2 + c_i + C_3$$

donde C_2 solo contiene paréntesis cerrados o es vacía. Nótese que en C_1 hay paréntesis abiertos sin cerrar suficientes para balancear a C_2 pues de lo contrario C no esatría balanceada (Si hubiera que utilizar a c_j para balancear a C_2, entonces c_i quedaría desbalanceado.

Sea

$$C' = C_1 + C_2 + c_j + c_i + C_3$$

C' contiene los mismos caracteres que C por tanto es una supersecuencia de A y B de la misma longitud de C.

El proceso de cambiar a c_j de lugar en la cadena puede verse como eliminar a c_j de C y luego agregarlo antes de c_i. Al eliminarlo solo se ha desbalanceado un paréntesis cerrado que estaba después de c_j en C, como sabemos que en C_1 hay paréntesis abiertos sin cerrar suficientes para balancear a C_2 entonces poniendo a c_j después de C_2 se mantiene el balance en la cadena.

5.3.2. Complejidad temporal

El análisis es análogo al propuesto en la complejidad temporal de la Solución 2, difiere solamente en los parámetros que se calculan en cada propuesta, cuyas actualizaciones en cada llamado recursivo se hacen con un costo de tiempo constante para ambos casos y por tanto no altera su complejidad.