Práctica 4 “Multiplicación encadenada de matrices”

Angel Zait Hernández López
No. Boleta: 2014080682
Análisis de Algoritmos, Grupo: 3CM4

2 de mayo de 2020

Resumen

Un algoritmo voraz (también conocido como ávido, devorador o goloso) es aquel que, para resolver un determinado problema, sigue una meta-heurística, consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima [2]. La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas [3].

Palabras clave: Algoritmo, voraz, programación, dinámica.

1. Introdución

Los algoritmos voraces suelen ser muy eficientes, sin embargo, existen problemas dónde no encuentran una solución óptima. Una alternativa es el uso de técnicas de programación dinámica.

1.1. Multiplicación de una secuencia de matrices

Sean A, B, C matrices, tal que $C = A \times B$. Esto solo es posible si la matriz A tiene dimensiones pq y la matriz B dimensiones $q \times r$, resultando una nueva matriz C co dimensiones $p \times r$.
Dado que la multiplicación de una secuencia de matrices es asociativa, es decir, se pueden agrupar en sub-secuencias de multiplicación por pares, sin cambiar el orden de estas, para operar, se tiene el problema de encontrar como asociar estas matrices para tener el menor número de operacioens (sumas y multiplicaciones) posibles.
Veremos un pequeño ejemplo con solo cuatro matrices, que son A, B, C y D. Los posibles cálculos de dichas matrices son las siguientes:

- $((A \times B) \times C) \times D$

- $(A \times (B \times C)) \times D$
- $A \times ((B \times C) \times D)$
- $A \times (B \times (C \times D))$
- $(A \times B) \times (C \times D)$

El producto de matrices tiene distintas aplicaciones, tales como:

- Facilitar la resolución de problemas matemáticos.
- Obtener cálculos financieros de manera óptima y ordenada.
- Solución para sitemas de ecuaciones con muchas variables.

2. Planteamiento del problema

Como podímos notar en la sección 1.1, teniendo una cantidad n de matrices, podemos encontrar m formas de multiplicar dichas matrices, pero una forma de multiplicar pude ser más costosa que otra, es decir, que al momento de realizar las operaciones, hacemos más proceso matemático en una a comparación con otra. Entonces, podemos ver que una multiplicación de matrices pude generar diferentes soluciones, pero lo que buscamos aquí, es realizar la menor cantidad de operaciones matemática para resolver un problema.
Para ello, utlizaremos dos tipos de algoritmos ya conocidos, que son el algoritmo voráz y la programación dinámica; recordandon un poco de ambos, el algoritmo voraz se encarga de hacer el mejor conjunto de soluciones al problema para poder dar la mejor respuesta o una de las mejores, en cambio, con la programación dinámica se hacen tablas o matrices de soluciones que conforme vamos avanzando para hallar la solución, este revisa los resultados anteriores para poder tener la solución más óptima del problema.
Entonces podemos resumir el problema en lo siguiente:
Dado que la multiplicación de una secuencia de matrices es asociativa, es decir, se pueden agrupar en sub-secuencias de parejas de matrices, sin cmabiar el orden de estas, para operar. Se tiene el problema de encotrar como asociar estas matrices para tener el menor número de operacioens (sumas y multiplicaciones) posibles.
En la siguiente sección, se redactará de que se tratan los algoritmos utilizados para resolver el problema.

3. Desarrollo de la práctica

El lenguaje que se utilizó para el desarrollo de esta práctica fue C++, ya que su velocidad es mayor a comparación de otros lenguajes, como lo es Python ó Java. Esto se tomó a consideración por el tipo de problema que se nos presenta, porque puede que exista un problema muy grande y se quiera aproximar a la mejor solución de una manera rápida. Además de que se optó usar clases, ya que ambos tipos de algoritmos usan las mismas entradas, por lo que es más fácil inicializar un objeto con sus atributos correspondientes.

3.1. Algoritmo voraz (Algoritmo Breedy)

Como dijimos en la sección anterior, el algoritmo voraz va a tratar de buscar la mejor solución de una forma un poco forzada, pero con este algoritmo, vamos a buscar una forma óptima, ayudándonos con dos listas, las cuales tendran las matrices a multiplicar, que serán ordenadas de distintas formas.
A estas listas las llamaremos $Lista00$ y $Lista01$, dónde $Lista00$ almacenará las matrices ordenadas de mayor a menor número de columnas y $Lista01$ estarán ordenadas de mayor a menor número de filas. Pongamos un pequeño ejemplo para poder entender un poco más esto. Supongamos que tenemos estas matrices con las siguientes dimensiones (Tabla 1).

Matrices				
Nombre	A_1	A_2	A_3	A_4
Dimensiones	3×6	6×9	9×2	2×7

Tabla 1: Conjunto de matrices

Con ayuda de estos datos, ordenaremos las matrices para poderlas guardar en la $Lista00$ y a su vez en la $Lista01$, siguiendo las condiciones, quedarían de la siguiente forma:

- $Lista00 \leftarrow \{A_2, A_4, A_1, A_3\}$
- $Lista01 \leftarrow \{A_3, A_2, A_1, A_4\}$

Después de tener ordenadas estás vamos a tomar el último elemento de la $Lista00$, el cual se llamará $matA$ y buscaremos en la $Lista01$ con que matriz se puede multiplicar, esta matriz la llamaremos $matB$. Cómo sabemos, $matA$ y $matB$ tienen un cierto numero de filas y columnas y nosotros para poder multiplicar las matrices se debe tener el mismo número de filas de una matriz y el mismo número de columnas en la otra matriz.
Tomemos de nuevo nuestro ejemplo, tomaremos la última matriz, en este caso es A_3 y buscaremos en la $Lista01$ la matriz que se pueda múltiplicar, que en este caso indagaremos para poder encontrar una matriz que su número de columnas coincida con el número de filas de A_3. Finalmente, podemos ver que las matrices A_3 y A_2 pueden multiplicarse. Ahora haremos un nuevo paso, después de esto, haremos un cálculo del número de operaciones que pueden realizar y esto se obtiene multiplicando el número de filas de A_2 por el número de columnas de A_2 por el número de columnas de A_3, en este ejemplo dará como resultado 108 operaciones a realzar.
Una vez realizada la operacióin, se para a remover las matrices A_2 y A_3 de ambas listas y se agregará una nueva, que llevará el nombre de B_1 y cuyas dimensiones serán el número de filas de A_2 y el número de columnas de A_3, es decir, que las dimensiones quedarían $B_1(6 \times 2)$, y esta nueva matriz se agregará a la $Lista00$ de manera ordenada, entonces, ambas listas quedarán así:

- $Lista00 \leftarrow \{A_4, A_1, B_1\}$
- $Lista01 \leftarrow \{A_1, A_4\}$

Buscamos de nuevo una matriz que multiplique a B_1, pero si no encontramos una matriz que multiplique a B_1 nos iremos al dato anterior de la lista que sería A_1, y se sigue el proceso hasta que la $Lista01$ esté vacía o bien, no encontremos otra matriz que se pueda multiplicar entre los elementos de la $Lista00$ y $Lista01$.
Acabado esto, ignoramos la $Lista01$ y solamente utilizaremos la $Lista00$ y buscaremos entre sus matrices cuales se pueden multiplicar entre si, tomando en cuenta el numero de filas y columnas, es decir, tomando nuevamente la $Lista00$ como está, B_1 se multiplicar con la matriz A_4, se eliminarían los elementos B_1 y A_4 y se generaría la nueva matriz con $B_2(6 \times 7)$, y se vuelve a verificar con qué matrices se puede multiplicar, hasta que la $Lista00$ solamente tenga un elemento y termina el algoritmo.
De nuevo recordando, este algoritmo no asegura que se tenga el resultado más óptimo, pero trata de acercarse a una buena solución, además de que va tratando de descartar poco a poco todas las matrices utilizadas. Para terminar, este algoritmo devuelve el número de operaciones que se pueden realizar con ese conjunto de matrices, este algoritmo se puede ver mejor en el algoritmo 1.

3.2. Programación Dinámica (Matrix-Chain-Orden)

En esta solución, el objetivo es encontrar una estructura para generar soluciones óptimas a subproblemas y después combinar las soluciones parciales que se genaran en las iteraciones pasadas. Trabajaremos ahora con un arreglos y dos matrices el cual llamaremos p al arreglo que guardaá las dimensiones de cada una de las matrices, la matriz m que almacenará el número de operaciones de las asociaciones óptimas y la matriz s que servirá para guardar las decisiones en cada iteración.
Tomando de nuevo las matrices de la tabla 1, vamos a crear nuestro arreglo p que contiene las dimensiones de las matrices, por lo que quedaría de la siguiente forma:

3	6	9	2	7

Tabla 2: Arreglo p

Una vez tiendo esto vamos a poner crear dos matrices (s y m), con el tamaño del arreglo p menos uno. A continuación, la diagonal principal de las matrices se iniciará con ceros. Y de ahora en adelante, trabajaremos con los valores de la matriz m y el arreglo s. Como recordaremos, en la programación dinámica usar las matrices para saber si la desición anterior es mejor o no que la actual.
Es por eso, que el la parte del algoritmo 2 en la linea doce a la linea dieciséis. En esta parte estamos tomando los valores anteriores que se han calculado del arreglo m, los valores del arreglo p y con esos valores, vamos a comparar si el nuevo calculo es mejor que el anterior, para poder guardarlo. esta parte podemos verla como la siguiente forma:

$$min\{m[i, j] + m[k + 1, j] + p_{i-1}p_kp_j\} \quad (1)$$

Dónde k toma valores entre i y j ($i \leq k \leq j$). Esto nos da a ver que vamos a estar recorriendo todas las posibilidades que hay al multiplicar las multiplicaciones entre si. Y el cual tomará los valores de k para guardarlas en s y así después poder obtener el número de operaciones óptimo para multiplicar las matrices y a su vez, imprimir la mejor manera de multiplicar dichas matrices.
Para la impresión del conjunto de multiplicaciones optimas, se utiliza el algoritmo 3, que se ayuda con la matriz s, para poder saber que matriz sigue en la impresión.

4. Pruebas

En esta sección se hicieron un par de pruebas con datos específicos, aquí podremos ver y comparar los algoritmos voraces y la programación dinámica en los problemas a resolver. Para mayor información de ejecución, se anexa a la práctica un archivo "README.txt"donde se puede leer las especificaciones de como compilar y correr el programa.
Para probar este algoritmo, se utilizaron varios conjuntos de matrices, los cuales, se mostrarán en la tabla 3:

Conjunto	**Matrices**					
1		$\mathrm{A}_1(50 \times 30)$		$\mathrm{A}_2(30 \times 20)$		$\mathrm{A}_3(20 \times 100)$
2	$A_1(10 \times 200)$	$A_2(200 \times 300)$	$A_3(300 \times 50)$	$A_4(50 \times 91)$	$A_5(90 \times 10)$	
3	$A_1(1 \times 2)$	$A_2(2 \times 3)$	$A_3(3 \times 4)$	$A_4(4 \times 5)$	$A_5(5 \times 6)$	$A_6(6 \times 7)$

Tabla 3: Conjuntos de prueba

En la tabla 3 se tienen 3 conjuntos distintos a los que se sometieron los Algoritmo Breedy y Matrix-Chain-Orden. Para poder ver el funcionamiento de cada uno, se hará una pequeña tabulación con los tiempos de que tanto se tardan ambos algoritmos. Va-

Conjunto 1			
Algoritmo	**Operaciones**	**Asociación**	**Tiempo (seg)**
Algoritmo Breedy	130'000	$(A_2 \times A_1) \times A_3$	0.004
Matrix-Chain-Orden	130'000	$((A_2 \times A_1) \times A_3)$	0.006

Tabla 4: Comparación del Conjunto 1

yamos uno por uno, en la parte del conjunto uno y como vemos en la tabla 4 y podemos notar que son solo 3 matrices y que ambos nos dan el mismo resultado, pero en el tiempo el Algoritmo Breedy es mucho más rápido que el otro, aunque sea por poco.
Para la tabla 5, se ve más clara la diferencia, ya que ahora son 5 matrices las que se van a multiplicar. Además de la diferencia de tiempos, que es considerable, las operaciones de uno a otro tienen una diferencia de 15'000 operaciones, hablando computacionalmente es demasiado proceso para cualquier computadora. Agregando que también el orden de multiplicar es de distinta forma. Recordando de nuevo el tipo de algoritmo que se utiliza el algoritmo voráz va a tratar de hacerlo como llegen los datos mientras que el otro verifica cual es el mejor.

Conjunto 2			
Algoritmo	**Operaciones**	**Asociación**	**Tiempo (seg)**
Algoritmo Breedy	815'000	$(((A_5 \times A_4) \times A_3) \times A_2) \times A_1$	0.016
Matrix-Chain-Orden	800'000	$(((A_1 \times A_2) \times A_3) \times (A_4 \times A_5))$	0.004

Tabla 5: Comparación del Conjunto 2

Conjunto 3			
Algoritmo	**Operaciones**	**Asociación**	**Tiempo (seg)**
Algoritmo Breedy	326	$((A_2 \times A_1) \times (A_4 \times A_3)) \times (A_6 \times A_5)$	0.016
Matrix-Chain-Orden	110	$(((((A_1 \times A_2) \times A_3) \times A_4) \times A_5) \times A_6)$	0.006

Tabla 6: Comparación del Conjunto 3

Por último, en la tabla 6, observamos lo mismo que en el conjunto pasado, el número de operaciones tiene una diferencia, pero no tan grande como la de la tabla 5, también, si lo notamos, tiene que ver con las dimensiones que tienen las matrices y el numero de ellas. Para finalizar, el tiempo, notamos que el algoritmo Matrix-Chain-Orden es más rapido que el Algoritmo Breddy, pero cuando son un conjunto de matrices muy reducido, puede que ambos den batalla al resolver un problema de multiplicación de matrices.
Para poder ver las pruebas en funcionamiento puede ejecutar el código, o bien, pude ver las figuras 1 a la 6.

5. Artículo "Very Fast Approximation of the Matrix Chain Product Problem"

Este artículo nos comparte que la solución al problema de relacionar y encontrar una manera óptima de multiplicar las matrices es por medio de una triangulación ṕtima de un polígono convexo, el cual fue mencionado por Hu and Shing [4]. Este algoritmo nos dice que dado un polígono $P = (v_0, v_1, ..., v_n)$ con pesos positivos asociados en cada vértice, lo divideremos en triángulos para que el costo total de partición sea lo más pequeño posible. El costo de una triangulación es la suma de los costos de todos los triángulos particionados. El costo de un triángulo, es el producto de los pesos en cada vértice del triángulo.
También nos explica como es que hacemos las divisiones en un polígono, en resumen, nos dice que el polígono va a tener un cierto numero de vérticies, entonces, para dividirlo en triángulos, nosotros vamos a posicionarnos en el primer vértice para saber que tan cercano está uno del otro, para así hacer cada vez más polígonos pequeños, que estos poco a poco van a ir formando un triángulo.
Otra cosa que nos menciona este articulo, es sobre el altoritmo de aproximación Chin-Hu-Sing, el cuál también es conocido como algoritmo CHS. En el ejemplo, nos dan cuatro vértices dńde v_m es el vertice más pequeño de un poligono convexo y que v_t es

un vertice con v_k y v_c como dos vecinos. Define el vertice v_t siendo “largo”si:

$$\frac{1}{v_k} + \frac{1}{v_c} > \frac{1}{v_t} + \frac{1}{v_m} \tag{2}$$

Además, en esta parte nos da dos pasos o condiciones importantes para la unión de vertices y división de un polígono:

1. Mientras haya un vertice largo v_t, corte v_t por el arco que conecta sus dos vecinos y elimine v_t del poligono (Figura 7); el arco de conección de los vecinos de v_t se le llama un *h-arc*

2. Si ninguno de los vertices es largo, entonches el vertice más pequeño es conectado a todos los otros vertices (Figura 8); los arcos obtenidos se llaman *fan-arcs*

También, se menciona otro algoritmo que es el algoritmo de aproximación paralela, y donde nos menciona que este es basado en el algoritmo CHS, solo que agrega y puntualiza un lema:
“Si un vertice v_t es cortado por un *h-arc* (v_k, v_c), entonces (v_k, v_c) es la candidato para guardar con v_t.” [1]

5.1. Algoritmo

A pesar de que no está descrito con detalle el algoritmo, este hace mención de todos los métodos en los que se basaron para poder hacer proceso, y se trata principalmente de lo siguiente:
La entrada es una seguencia de enteros $(d_0, d_1, ..., d_n)$ de dimensiones de matrices. Un orden de multiplicación corresponde a un método de poner $n-1$ pares de paréntesis anidados alrededor de n matrices. La salida será un arreglo llamado $'Brackets' \leftarrow (B_1, ..., B_{n-1})$ que contendrá la posicin de todos los pares de parentecis que aparecen en un orden óptimo de la multiplicación de matrices. Ahora, para cada i, $1 \leq o \leq n-1$, $B_i \leftarrow (j, t)$ solamente si la expresión $(M_{j+1} \times M_{j+2} \times \cdots \times M_t)$ está presente en este orden óptimo; es decir, hay un parentesis alredeor de M_{j+1} y M_t.

5.2. Comparación de algoritmos

La propuesta que nos hacen en este artículo es llamativa, ya que la complejidad puede que sea un poco menor que las otras, al igual que la disminusión de tiempo. Considero que, a pesar de la época que era, este método puede que sea un poco costoso en consumo de memoria, ya que necestia hacer muchos recortes de poligonos y considerar varios casos que se den, ya que, como podemos ver, este algoritmo se basa en muchos otros métodos.
Además la implementación del algoritmo puede ser un poco confusa, por lo mismo de la dificultad de entender el como generar un polígono irregular, a pesar de que los valores son enteros positivos, este a su vez puede llegar a ser pesado para la computadora el hacer demasiados cálculos y crear cada vez un recorte y recordar todas las vértices que se fueron uniendo, siendo el peor caso de dicho método.

Bien puede hacer buena competencia al método Matrix-Chain-Orden, pues este reduce algunos pasos para poder encontrar dicha multiplicación de matrices, ahora, si lo comparamos con el Algoritmo Breedy, es demasiado eficiente la creación de poligonso convexos, porque si ponemos un número muy grande de matrices que se necestien multiplicar, el Algoritmo Voraz hará mucho más operaciones y el tiempo de respuesta será tardada a lado de este algoritmo de polígonos convexos.
Es buena opción para poder reemplazar ambos métodos, pero no nos dice en concreto como es el algoritmo este artículo, y puede que, a pesar de que se leé sencillo, la implementación pude ser un poco tediosa y hasta confusa por los tipos de conceptos que se hablan.

6. Conclusiones

Como podemos observar, siempre habrá más de una solución a un problema, solo es cuestión de pensar e informarse. La programación es libre, siempre y cuando se cumplan los criterios.
De nueva cuenta, observamos que el algoritmo voraz puede hacerle competencia a los métodos de programación dinámica en cuestión de tiempo y solución cuando hay pocos factores a analizar, en este caso, que haya un número muy pequeño de matrices, como fue el caso del conjunto 1. Pero cada vez que vamos aumentando más la complejidad del problema, el algoritmo voraz se ve nuevamente acorralado por la programación dinámica.
Como se ha comentado anterior mente, es porque los métodos dinámicos siempre tiene esa pequeña memoria de si la solución que va obteniendo es buena o no. En cambio el voraz, lo primero que se encuentra lo va a tratar de hacer y así solo acumular la solución sin importar si es la mejor o no.
Finalmente, tomando un poco en cuenta el algoritmo del articulo "Very Fast Approximation of the Matrix Chain Product Problem", puede que exitan mejor formas de solucionar el mismo problema, pero de nueva cuenta, hay que investigar un poco más, además de que hasta cierto punto es un poco complejo, hablando en la implementación del método.

Referencias

[1] CZUMAJ, A. Very fast approximation of the matrix chain product. 71 – 79.

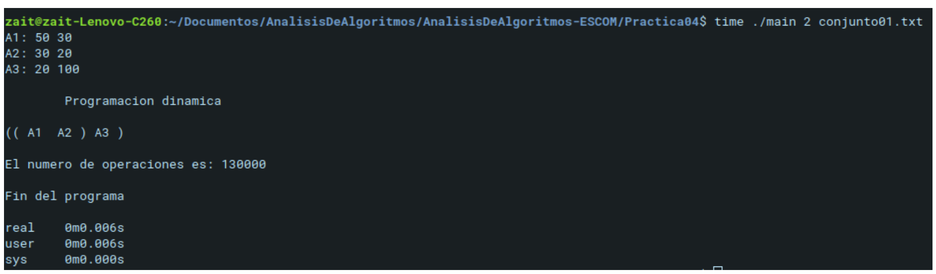
[2] ECURED. Algoritmos voraz, https://www.ecured.cu/algoritmos_voraz.

[3] ECURED. Programación dinámica, https://www.ecured.cu/programación_dinámica.

[4] HU, T. C., AND SHING, M. T. Some theorems about matrix multiplications. 28 – 35.

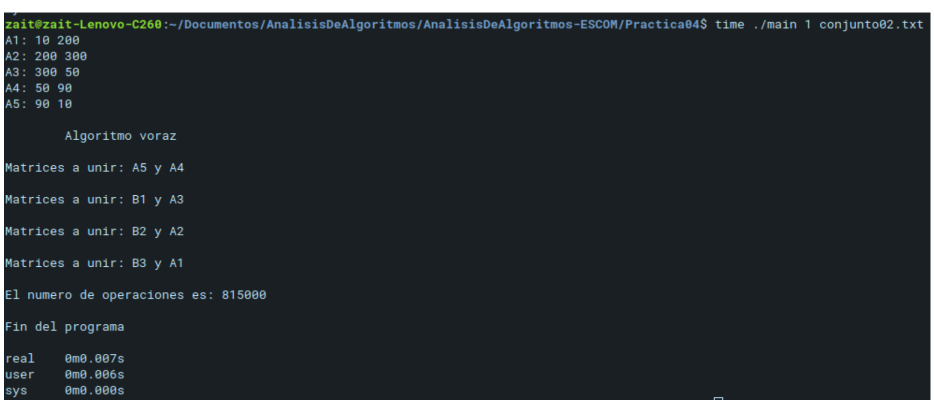
```
zait@zait-Lenovo-C260:~/Documentos/AnalisisDeAlgoritmos/AnalisisDeAlgoritmos-ESCOM/Practica04$ time ./main 1 conjunto01.txt
A1: 50 30
A2: 30 20
A3: 20 100

        Algoritmo voraz

Matrices a unir: A2 y A1

Matrices a unir: B1 y A3

El numero de operaciones es: 130000

Fin del programa

real    0m0.006s
user    0m0.006s
sys     0m0.000s
```

Figura 1: Prueba de algoritmo voraz con el conjunto 1



```
zait@zait-Lenovo-C260:~/Documentos/AnalisisDeAlgoritmos/AnalisisDeAlgoritmos-ESCOM/Practica04$ time ./main 2 conjunto01.txt
A1: 50 30
A2: 30 20
A3: 20 100

        Programacion dinamica

(( A1  A2 ) A3 )

El numero de operaciones es: 130000

Fin del programa

real    0m0.006s
user    0m0.006s
sys     0m0.000s
```

Figura 2: Prueba de programación dinámica con el conjunto 1



```
zait@zait-Lenovo-C260:~/Documentos/AnalisisDeAlgoritmos/AnalisisDeAlgoritmos-ESCOM/Practica04$ time ./main 1 conjunto02.txt
A1: 10 200
A2: 200 300
A3: 300 50
A4: 50 90
A5: 90 10

        Algoritmo voraz

Matrices a unir: A5 y A4

Matrices a unir: B1 y A3

Matrices a unir: B2 y A2

Matrices a unir: B3 y A1

El numero de operaciones es: 815000

Fin del programa

real    0m0.007s
user    0m0.006s
sys     0m0.000s
```

Figura 3: Prueba de algoritmo voraz con el conjunto 2

Figura 4: Prueba de programación dinámica con el conjunto 2

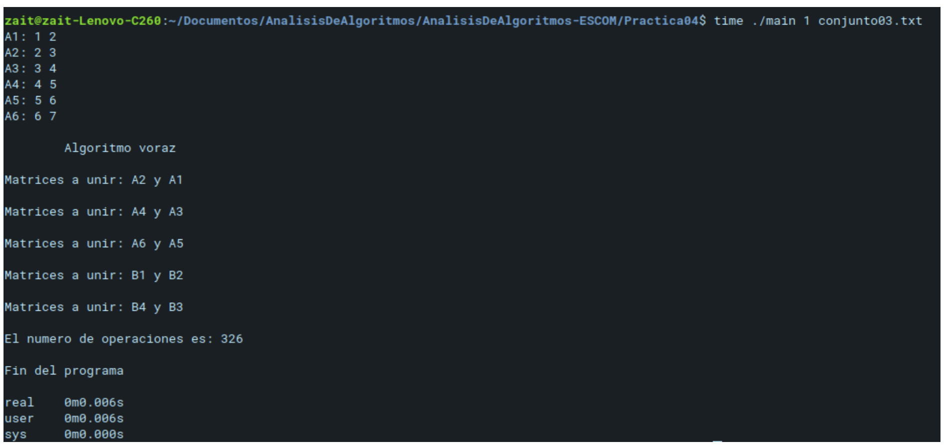


Figura 5: Prueba de algoritmo voraz con el conjunto 3

Figura 6: Prueba de programación dinámica con el conjunto 3

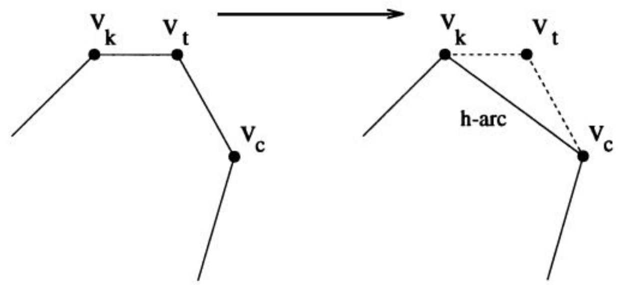


Figura 7: Unión de vértices vecinos

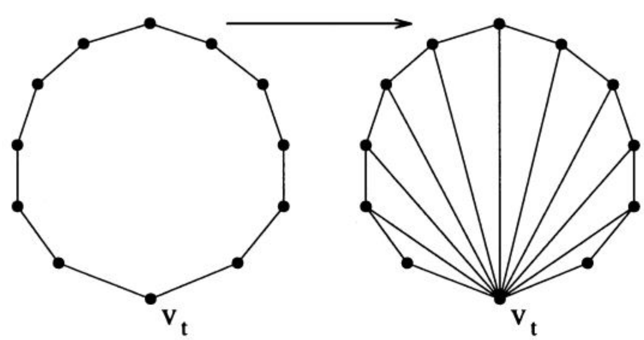


Figura 8: Unión de todos los vértices

Algorithm 1: Algoritmo voráz. Algoritmo Breedy

Data: $Lista00$: lista de matrices ordenadas de mayor a menor número de columnas.
Data: $Lista01$: lista de matrices ordenadas de mayor a menor número de filas.
Result: $numOp$: Número de operaciones al multiplicar los datos.

begin
    /*
        Las variables $matA$ y $matB$ son matrices con un numero de filas y columnas (dimensiones)
        $nombre$ que puede ser A_i o B_j
        $numFilas \leftarrow$ nḿero de filas
        $numColum \leftarrow$ número de columnas
    */
    $i \leftarrow 0$
    while *Lista01 no esté vacía ó no haya elementos que se puedan multiplicar entre las matrices de Lista00 y la Lista01* **do**
        $matA \leftarrow$ el último elemento de la $Lista00$
        $matB \leftarrow$ el primer elemento de la $Lista01$ que se pueda multiplicar con $matA$
        if $matB.numColum == matA.numFilas$ **then**
            $i \leftarrow i + 1$
            $numOp \leftarrow numOP + (matB.numFilas * matB.numColum * matA.numColum)$
            Se crea una nueva matriz llamada B_i con las dimenciones:
            $B_i.numFilas \leftarrow matB.numFilas$
            $B_i.numColum \leftarrow matA.numColum$
            Se eliminan de ambas listas los elementos $matA$ y $matB$
            Se agrega la matriz B_i a la $Lista00$ de forma que quede ordenada con la condición de $Lista00$
        else
            El apuntador del final de la $Lista00$ se recorre un elemento antes, para tomar el siguiente elemento de la lista.
    while *Lista00 tenga solo un elemento* **do**
        $matA \leftarrow$ el último elemento de la $Lista00$
        $matB \leftarrow$ el primer elemento de la $Lista00$ que se pueda multiplicar con $matA$
        if $matB.numColum == matA.numFilas$ **then**
            $i \leftarrow i + 1$
            $numOp \leftarrow numOP + (matB.numFilas * matB.numColum * matA.numColum)$
        else
            $i \leftarrow i + 1$
            $numOp \leftarrow numOP + (matA.numFilas * matA.numColum * matB.numColum)$
        Se crea una nueva matriz llamada B_i con las dimenciones:
        $B_i.numFilas \leftarrow matB.numFilas$
        $B_i.numColum \leftarrow matA.numColum$
        Se eliminan de ambas listas los elementos $matA$ y $matB$
        Se agrega la matriz B_i a la $Lista00$ de forma que quede ordenada con la condición de $Lista00$
    return $numOp$
end

Algorithm 2: Programación dinámica. Matrix-Chain-Orden

Data: p: Arreglo donde se guardan las dimensiones de cada una de las matrices.
Result: s: Matriz donde se guardaran las decisiones en cada iteración.
Result: m: Matriz donde se almacena el número de operaciones de las asociaciones óptimas

begin
  $n \leftarrow p.length - 1$
  let $m[1...n, 1...n]$
  let $s[1...n, 1...n]$
  for $i \leftarrow 1$ *to* n **do**
    $m[i, i] \leftarrow 0$
    $s[i, i] \leftarrow 0$
  for $l \leftarrow 2$ *to* n **do**
    for $i \leftarrow 1$ *to* $n - l + 1$ **do**
      $j \leftarrow i + l - 1$
      $m[i, j] \leftarrow \infty$
      for $k \leftarrow i$ *to* $j - 1$ **do**
        $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$
        if $q < m[i, j]$ **then**
          $m[i, j] \leftarrow q$
          $s[i, j] \leftarrow k$
  return s y m
end

Algorithm 3: Print-Optimal-Parens

Data: s: Matriz donde se guardaron las decisiones en cada iteración.
Data: i: Entero que representa el índice.
Data: j: Entero que representa el índice.

Print-Optimal-Parens(s,i,j)
begin
  if $i == j$ **then**
    Imprimir “A”$_i$
  else
    Imprimir “(”
    Print-Optimal-Parens(s,i,$s[i, j]$)
    Print-Optimal-Parens(s,$s[i, j] + 1$,j)
    Imprimir “)”
end