Informe Proyecto

FADA

Escuela de Ingeniería de Sistemas y Computación

Estudiantes:

Juan Sebastian Velasquez 1744936

Cristian Llanos Alvarez 1943852

Profesor

Jesus Alexander Aranda Bueno

Universidad del valle

Santiago de Cali

2022

Tabla de Contenido

Optimizando los tiempos de producción en líneas de ensamblaje

Caso concreto: Optimizar el número de total de horas para ensamblar un producto usando dos líneas de ensamblaje

Definición del problema como uno de optimización

- Entradas:
  - N actividades en 2 líneas de ensamblaje teniendo
  - <a1,a2,a3 … aN> donde ai es el número de horas que necesita la línea ‘a’ para realizar la actividad i.
  - Teniendo <b1,b2,b3,...bN> donde bi es el número de horas que necesita la línea ‘b’ para realizar la actividad i.
  - Se tiene también los tiempos de transferencia <ba1, ba2, ba3, …baN-1> donde bai es el tiempo para pasar de la línea ‘b’ a la línea ‘a’ asumiendo que ya se realizó la actividad i.
  - De manera análoga se tiene <ab1, ab2, ab2, …abN-1> donde abi es el tiempo para pasar de una línea de la línea ‘a’ a la línea ‘b’ asumiendo que ya se realizó la actividad i.
- Salidas:
  - t : Tiempo Total en horas que se demora la línea de producción al escoger el camino más óptimo a través de las dos líneas de ensamblaje.
  - <x1, x2, x3, …xN> donde xi vale ‘a’ o ‘b’ indicando que la actividad i se realizó en la línea de ensamblaje ‘a’ o ‘b’ y de esta forma se representa el camino más eficiente. Considerando también los tiempos de transferencia.
- Se busca <u>minimizar</u> el número total de horas para ensamblar un producto usando dos líneas de ensamblaje.

$$\sum_{i=1}^{N} xi \text{ sea mínimo}$$

Definición recursiva de la solución

En principio, por facilidad se considera que la línea a→0 y la línea b→1, entonces:

lineasEnsamblaje(i,N) es el problemas cuya solución es <x1,x2,x3,...xN> sabiendo que i puede obtener solo 2 valores (0,1) porque se tienen solo 2 línea de ensamblaje.

Se tiene entonces que <x1,x2…,xN-1> es la solución a partir de 2 subproblemas:

- lineasEnsamblaje(0, N-1) y
- líneasEnsamblaje(1, N-1)

Esto quiere decir que los subproblemas cuando hay N-1 actividades en la línea 0→a y la línea 1→b conforman la solución óptima <x1,x2…,xN-1>.

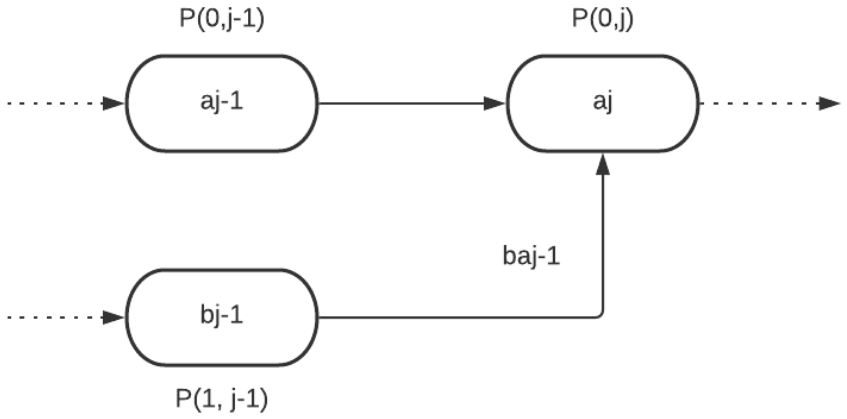
Definición recursiva del costo de la solución

El costo de la solución óptima está dado por el tiempo, para poder definir el costo de la solución óptima se necesita aclarar algunos comportamientos.

¿Cómo se computa el tiempo mínimo de pasar sobre una línea de ensamblaje?

Todos los caminos posibles para ir desde un punto de inicio a través de la línea de ensamblaje aj.

Tenemos dos opciones:



Opción 1 para llegar a aj: Desde el anterior punto de ensamblaje de la misma línea ‘a’ (aj-1→aj)

Opción 2 para llegar a aj: Desde el anterior punto de ensamblaje de la línea ‘b’ (bj-1→baj-1→aj), teniendo en cuenta que hay un costo de transferencia baj-1.

Generalización:

Una solución óptima del problema “Camino más rápido para llegar a ‘aj’ “ contiene una solución óptima a los subproblemas:

- “Camino más rápido desde el anterior punto de ensamblaje de la mísma línea (aj-1→aj)”
- “Camino más rápido desde el anterior punto de ensamblaje desde la otra línea (bj-1→baj-1→aj)”

La anterior generalización nos ayuda a construir una solución óptima del problema, a través de soluciones óptimas de subproblemas.

Con esto podemos definir una función objetivo:

- TiempoTotal = min (T(0,n) , T(1,n))

TiempoTotal → Menor tiempo posible total para ensamblar el producto

T(i,j) → El menor tiempo posible para llegar a un punto de ensamblaje P(i,j) desde un punto de partida.Siendo i la línea de ensamblaje y j el número de actividad.

El tamaño de la matriz para encontrar el valor de la solución óptima tendría una dimensión de 2xN.

También podemos definir una matriz auxiliar a partir de la cual podremos calcular la solución óptima:

L(i,j) → El número de línea (0 o 1) donde el punto P(i,j-1) es usado para llegar a P(i,j). Siendo i la línea de ensamblaje y j el número de actividad.

El tamaño de la matriz para encontrar la solución óptima tendría una dimensión de 2xN.

Casos base:

El punto de inicio para realizar la actividad 1 en la línea 0 o en la línea 1.

- T(0,1) = a1
- T(1,1) = b1

Con j = 1 e i= 0 - 1

Caso general:

Cuando 2<= j <= n e i = 0 - 1

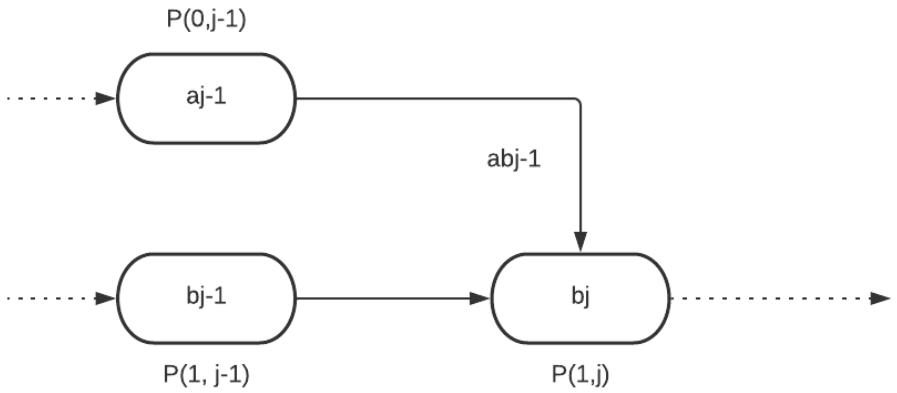
Tomando como referencia de nuevo las dos opciones:

1. “Camino más rápido desde el anterior punto de ensamblaje de la mísma línea (aj-1→aj)”. Daría un costo de = T(0,j-1) + a_j
2. “Camino más rápido desde el anterior punto de ensamblaje desde la otra línea (bj-1→baj-1→aj)” = T(1,j-1) + ba_{j-1}+a_j

De lo anterior podemos concluir que:

T(0,j) = min (T(0,j-1) + a_j , T(1,j-1) + ba_{j-1}+a_j)

De manera análoga se puede definir el camino más rápido para llegar al punto P1,j:



De nuevo dos opciones para llegar a bj:

1. Desde la misma línea: bj-1 → bj. El costo es = T(1, j-1) + b_j
2. Desde la otra línea con un costo de transferencia: aj-1 → abj-1 → bj. El costo es = T(0,j-1) + ab_{j-1} + b_j

Función recursiva del costo de la solución:

1era forma, separado por línea de ensamblaje:

Línea de ensamblaje 0→a, 0 constante, j variable.

T(0,j) =
{
aj si j=1
min (T(0,j-1) + a_j , T(1,j-1) + ba_{j-1}+a_j) si 1≤ j ≤ n
}
Línea de ensamblaje 1→b, 1 constante, j variable.
T(1,j) =
{
bj si j=1
min (T(1,j-1) + b_j , T(0,j-1) + ab_{j-1}+b_j) si $1 \leq$ j ≤ n
}
2da forma, todo junto en una definición:
T(i,j) =
{
aj si j=1 & i=0
bj si j=1 & i=1
min (T(0,j-1) + a_j , T(1,j-1) + ba_{j-1}+a_j) si $1 \leq$ j ≤ n & i=0

min (T(1,j-1) + b_j , T(0,j-1) + ab_{j-1}+b_j) si $1 \leq$ j ≤ n & i=1
}

Para computar el valor de la solución óptima se usa bottom-up, encontrando primero soluciones óptimas a subproblemas. Para todo j>=1, cada valor de T(i,j) depende solo de los valores de T(0,j-1) y T(1,j-1).

Se pueden computar los valores de T(i,j) en una matriz de la siguiente manera:

Se recuerda que todo el análisis se hizo indexando desde 0, por lo tanto la siguiente matriz se comporta de la misma forma. Como indexamos desde 0, no se va hasta N sino hasta N-1

	0	1	2	…	N-1
T(0,j)	T(0,0)	T(0,1)	T(0,2)	…	T(0,N-1)
T(1,0)	T(1,0)	T(1,1)	T(1,3)	…	T(1,N-1)

---------->

Se itera con j desde 1 hasta N

De esta forma se evidencia cómo funciona la programación dinámica, no se tiene que repetir la computación de valores que ya se han computado antes y almacenado en la matriz.

Ejemplo

N=3

Línea a: 2,2,6

Línea b: 3,5,3

Cambios ab: 1,1

Cambios ba: 4,2

T→ Matriz de costo, tiempo total → dimensión 2x3

	Actividad 0	Actividad 1	Actividad 2
T(0,j)	2	4	10
T(1,j)	3	8	8

T = min (T0(n), T1(n)) = min (10,8) = 8

Lo anterior indica que la producción del producto termina en la línea ‘b’ → 1.

L→ Matriz auxiliar, para encontrar la solución óptima→ dimensión 2x3

	Actividad 0	Actividad 1	Actividad 2
L(0,j)	0	0	0
L(1,j)	1	1	0

Pseudocódigo para el llenado de matrices

En este pseudocódigo se indexa desde 0, como en toda la explicación hasta ahora. En principio se muestra la función encargada de llenar las matrices T y L. Luego, esta función sirve como auxiliar para otra función que se encarga de encontrar el valor de la solución óptima y la solución óptima.

llenarMatrices:

```
1. llenarMatrices(n,a,b,ab,ba)
2.      T = [ ]
3.      L = [ ]
4.      T[0,0] = a[0]
5.      T[1,0] = b [0]
6.      L[0,0] = 0
7.      L[1,0] = 1
8.      for ( j from 1 to n-1 ) do
9.              if T[0,j-1] +  a[j] <= T[1, j-1] + ba[j-1] + a[j] do
10.                     T[0,j] = T[0, j-1] + a[j]
11.                     L[0,j] = 0
12.             else do
13.                     T[0,j] = T[1, j-1] + ba[j-1] + a[j]
14.                     L[0,j] = 1
15.             if T[1,j-1] +  b[j] <= T[0, j-1] + ab[j-1] + b[j] do
16.                     T[1,j] = T[1,j-1] +  b[j]
17.                     L[1,j] = 1
18.             else do
19.                     T[1,j] = T[0, j-1] + ab[j-1] + b[j]
20.                     L[1,j] = 0
21.     return T,L
```

Complejidad llenarMatrices()

La operación más costosa en el algoritmo, es el for loop por medio del cual llenamos las matrices, este for va desde 1 hasta n-1, se toma en cuenta que las operaciones dentro del loop son constantes ya que son accesos y asignaciones a posiciones en el arrays.

$$\sum_{j=1}^{n-1} c \;=\; c(n-1) \;\rightarrow \mathrm{O(n)}$$

Se tiene entonces una complejidad lineal para este algoritmo de llenado de matrices.

solucionesOptimas:

Este algoritmo toma las matrices dadas por el algoritmo llenarMatrices(), y a partir de ellas encuentra el valor de la solución óptima y la solución óptima.

```
1.solucionesOptimas(T,L,n)
2.      time = 0
3.      finalLine = ‘ ’
4.      lines = [ ]
5.      if (T[0,n-1] <= T[1,n-1]) do
6.              time = time + T[0,n-1]
7.              finalLine = 0
```

```
8.      else do
9.              time = time + T[1,n-1]
10.             finalLine = 1
11.     lines.append(finalLine)
12.     for( j from n-1 to 0 ) do
13.             if( j == n-1 ) do
14.                     finalLine = L[finalLine][j]
15.             else do
16.                     finalLine = L[finalLine][j]
17.                     lines.append(finalLine)
18.     reverse(lines)
19.     transformAB(lines)
20.     return time, lines
```

En este algoritmo se usan dos funciones auxiliares, reverse y tranformAB, la primera se usa para revertir el order de la solución óptima, puesto que en el loop se va agregando al array el camino escogido desde el final hasta el inicio, y lo queremos de principio a fin. La segunda es para reconvertir los valores 1→b y 0→a, ya que hace bastantes pasos atras se decidió cambiar de nomenclatura por cuestión de facilidad. Ambas funciones auxiliares no afectan a la complejidad total, puesto que también son O(n).

La complejidad del algoritmo solucionesOptimas() es de O(n) puesto que solo recorre una vez un loop for desde n-1 hasta 0. El resto de operaciones tienen una complejidad constante y las funciones auxiliares que se usan ya se explicaron. Por lo tanto podemos concluir que este algoritmo tiene una complejidad lineal.

Entonces, tomando en cuenta las dos funciones principales para encontrar las matrices y para encontrar las soluciones. Tenemos una complejidad de O(n) + O(n) cuyo valor final viene siendo la misma cota superior O(n).

Por eso podemos decir que la solución planteada para el problema de ‘’Optimizar el número de total de horas para ensamblar un producto usando dos líneas de ensamblaje’’ usando programación dinámica tiene una complejidad lineal. Si se hubiese aplicado una solución por fuerza bruta de manera recursiva, se hubiese podido llegar a una complejidad exponencial de $O(2^n)$. Pero almacenando los valores de subproblemas, se logra una reducción bastante considerable en la eficiencia del algoritmo hasta O(n).

Implementación

Código en repositorio de github:
https://github.com/Odzen/ProyectoFinal_FADA/tree/main/code/dinamica

Características del hardware usado

Procesador: AMD A10-9620P RADEON R5, 10 COMPUTE CORES 4C+6G 2.50 GHz

RAM : 12,0 GB (11,5 GB usable)

Almacenamiento : 1TB disco rigido y 250gb estado solido

Características del software empleado

Sistema operativo : Windows 10

IDE : Spyder (Anaconda 3)

Relación entre los algoritmos y su implementación

En la carpeta code/dinamica se pueden encontrar los archivos que se usaron para la implementación, usando el lenguaje de programación python en su versión 3.9.7. Dentro de esta carpeta se encuentran 5 archivos python que permitieron realizar la implementación y desarrollo de lo requerido para el proyecto.

- dinamica.py : Es el archivo principal, puesto que en él es donde se implementa el pseudocódigo anteriormente mostrado, primero tenemos la función solve que recibe un n, y 4 arreglos (a,b,ab,ba). Este método es el que luego llamaremos en el main, y lo que hace simplemente es llamar otros dos métodos: llenarMatrices(n,a,b,ab,ba) y lo que retorna este método lo pasa luego como argumento para el método solucionesOptimas(T,L,n). Este último método retorna tres elementos, que es lo que pide la salida del problema, n que representa las actividades, time que representa el tiempo minimizado o valor de la solución óptima y lines que representa un array en donde está una secuencia de a-b líneas que dan las solución óptima. Hay otro método que se usa como auxiliar en solucionesOptimas que se llama transformAB(lines), este recibe una solución óptima dada en 1's y 0's y la transforma a b's y a's respectivamente.
- inOut.py: Se tiene un archivo que se encarga de leer entradas y escribir archivos de texto. Este archivo fue dado por el monitor y solo se le hicieron mínimos cambios en las rutas y en que ahora la función de lectura y escritura reciben argumentos. Las rutas para los archivos de entrada es: Pruebas/Test_in_files y para los archivos de salida están en: Pruebas/outDinamica. El archivo outi.txt corresponde a la entrada ini.txt, es decir por ejemplo el archivo out5.txt es el archivo que corresponde a la salida del algoritmo recibiendo como entrada los valores dados por el archivo in5.txt. De manera análoga funciona Pruebas/outVoraz.
- mainDinamica.py: Este archivo se encarga importar las funciones de los anteriores archivos para poder recibir una entrada y dar una salida en el formato requerido, tiene la función main() que lee un archivo, luego llama al método solve de dinamica.py y lo que retorna lo escribe en un nuevo archivo de texto out.txt.

- test_creation.py: Este es un archivo complementario que se decide crear con el sentido de crear pruebas automáticas con el formato requerido en archivos de texto, aunque la mayoría de pruebas fueron creadas y probadas a mano, este archivo fue útil para ver ciertos comportamientos de los algoritmos y para poder graficar el comportamiento. Principalmente lo que hace es que escoge una cota menor mayor que se puede modificar luego para valores de a y b escoge números random entre estas dos cotas y para valores de ab y ba escoge números random entre la cota menor y la cotaMayor/2. Cuando ya tiene calculado y almacenado todos los valores entonces lo escribe en el formato requerido en un archivo texto para luego ser leído por algunos de los algoritmos.
- plotting_py.py: Archivo para realizar gráficas (actividades vs times) que representan el comportamiento de un algoritmo calculando el tiempo. Para esto se usa la librería time y pylab.Tenemos dos funciones.
    - Función plotting_incremental: Recibe un número de actividades que pasarán por las dos líneas de ensamblaje, e itera desde 1 hasta número de actividades, con la ayuda del test_creation.py. Ejemplo si el número de actividades es igual a 10, creará 10 pruebas la primera con 1 actividad, la segunda con 2 actividades etc..A partir de las actividades y el tiempo en cada iteración se crea un gráfico y se cálcula un tiempo promedio.
    - Función plotting_redundante : Recibe un numeroPrueba y un n_veces, el cual es el número de prueba que se va a ejecutar, de las pruebas ya existentes en el directorio de pruebas, archivos in.txt. A partir del tiempo y la iteración en la que se ejecuta un prueba se arma un gráfico y se saca un tiempo promedio.

Aspectos propios de las estructuras de datos usadas y conceptos propios de la implementación de los algoritmos

Las estructuras de datos que más se usaron fueron principalmente los arreglos o arrays, para poder gestionar la información. En python estas estructuras de datos son consideradas listas, pero en general, para poder indexar, agregar o asignar algún elemento a una posición de una lista, tiene un costo constante. Además de que también en algunos archivos se usa la librería numpy de python que es optimizada para el manejo de arreglos y matrices, el rendimiento de las estructuras de datos numpy es mucho mejor que las listas en términos de tamaño ocupado en memoria, velocidad y funcionalidad.

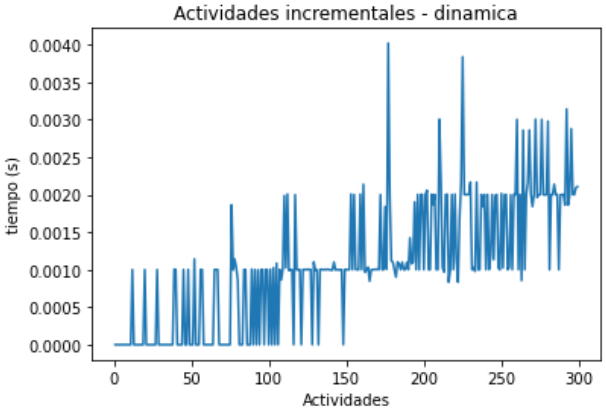
Gráficas - Pruebas

Para poder analizar el comportamiento real del algoritmo se realizaron algunas pruebas de las cuales se escogen del directori Pruebas/Test_in_files los archivos in2, in3, in10 y in12. Porque se cree que se prestan mejor al análisis no solo del algoritmo de dinámica sino también para el algoritmo voraz.

Los algoritmos anteriormente mencionados se utilizan en el archivo plotting_py.py por medio de la función redundante_test, que nos arroja un representación gráfica de la función dinamica.solve () y mide su tiempo medio de ejecución. Este redundate_test nos sirve para

comparar el algoritmo por dinámica y por voraz, algo que haremos más adelante en el informe.

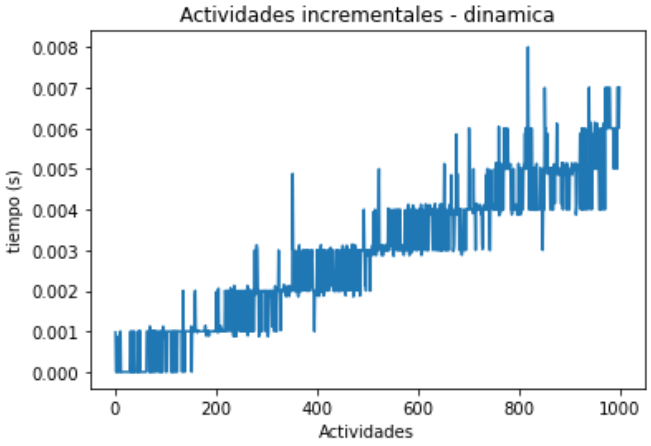
Otra forma de analizar el comportamiento del algoritmo con programación dinámica es por medio de la función incremental_test que ya explicamos anteriormente. Ejecutándola nos da el siguiente resultado:



Con una entrada de 300 actividades y un tiempo medio total de 0.0010561201484705692.

En la gráfica se puede observar el comportamiento lineal que tiene el algoritmo con dinámica. De esta forma se puede comprobar que la complejidad teórica O(n) no está tan lejos del comportamiento real de la solución.

Y con 1000 actividades tenemos los siguientes resultados:



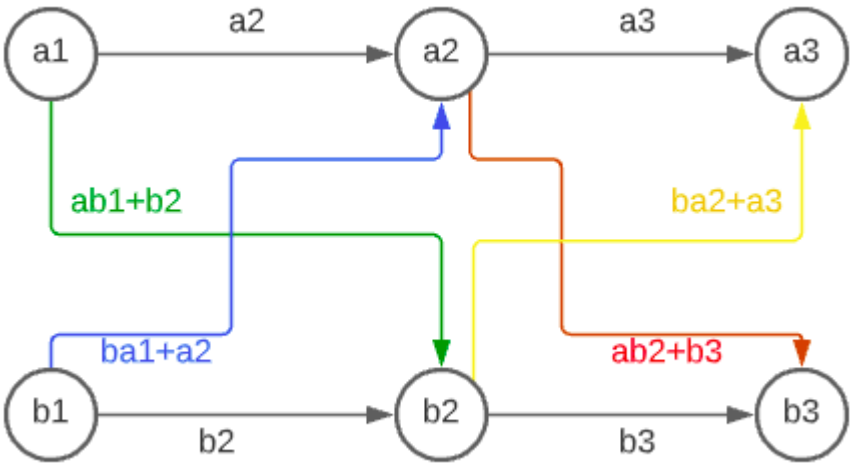
Se puede observar un comportamiento incluso más lineal que cuando la entrada fue 300, con un tiempo promedio de 0.0028935831946295663. Vemos entonces que a mayor n mayor es el tiempo promedio. Esto quiere decir que entre las dos variables existe una relación lineal incremental.

Programación Voraz

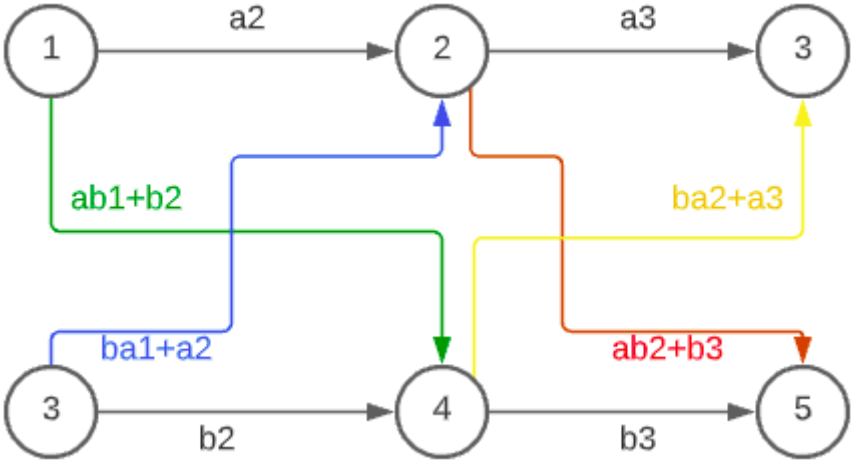
Para la realización del algoritmo voraz, se utilizó como referencia el algoritmo de Dijkstra, el cual resuelve el camino más corto entre dos vértices de un grafo ponderado, este algoritmo tiene complejidad $O(n^2)$.

Nuestro algoritmo recibe como entradas, una lista a y una lista b las cuales contienen el tiempo de duración de las actividades en las cadenas a y b respectivamente y una matriz de pesos mínimos de un grafo dirigido y ponderado, donde los vértices están numerados desde 1 hasta la suma de las longitudes de a y b, los vértices son la representación de las actividades de cada cadena y el peso de cada arista viene dado por el tiempo de duración de la siguiente actividad en la cadena si se mantiene en la misma cadena o por el tiempo que tarda en cambiar a la siguiente actividad en la otra cadena mas el tiempo de duración de dicha actividad como se muestra en el ejemplo de la siguiente figura. Para la dirección de las aristas se tiene en cuenta las restricciones del problema.

Modelo de grafo para ejemplo de 3 actividades



Modelo de grafo para ejemplo de 3 actividades, donde vemos las actividades representadas por vértices de 1 a n



Matriz de pesos mínimos de un grafo

para ejemplo de 3 actividades por línea de ensamblaje, donde el símbolo infinito representa que no hay aristas que conectan dos vértices entre sí:

vertice	a1	a2	a3	b1	b2	b3
a1	∞	**a2**	∞	∞	**ab1+b2**	∞
a2	∞	∞	**a3**	∞	∞	**ab1+b3**
a3	∞	∞	∞	∞	∞	∞
b1	∞	**ba1+a2**	∞	∞	**b2**	∞
b2	∞	∞	**ba1+a3**	∞	∞	**b3**
b3	∞	∞	∞	∞	∞	∞

Pseudocódigo voraz

solucionVoraz(actividades a, actividades b, matrizPesoMinimo[ ][ ]):

1. Inicio = verticeInicio(a,b)//Indica la actividad de menor peso entre a[1] y b[1]
2. tiempoTotalEsamble=inicio
3. ruta = { }
4. solucion = { }
5. actividadesTotal = { }
6. for i = to a.size

```
7.          actividadesTotal añadir a[i]
8.        for i = to b.size
9.          actividadesTotal añadir b[i]
10.   verticesCambio ={ }
11.       for i = to actividadesTotal.size
12.       verticesCambio=i
13.   vertices = { }
14.       for i = to actividadesTotal.size
15.       vertices=i
16.    division = verticesCambio.size/2
17.    ruta añadir posicionInicia
18.      if(inicio = b[1])
19.         posicionInicia = veticesCambio.size/2
20.         eliminar elemento verticesCambio[posicionInicia]
21.         eliminar elemento verticesCambio[1]
22.      else
23.         posicionInicia = 0
24.         eliminar elemento verticesCambio[posicionInicia]
25.         eliminar elemento verticesCambio[division]
26. pv
27.    while to verticesCambio este vacio
28.       pv=posicionInicia
29.       for i = to actividadesTotal.size
30.          solucion añadir matriz[ pv ][ i ]
31.       c = indice de minimo(solucion)
32.       e = indice de verticesCambio[c] en verticesCambio
33.       ruta añadir c
34.       division = verticesCambio.size/2
35.       if (e>division)
36.          eliminar elemento verticesCambio[e]
37.          eliminar elemento verticesCambio[0]
38.       else
39.          eliminar elemento verticesCambio[e]
40.          eliminar elemento verticesCambio[division]
41.    tiempoTotalEsamble += valor minimo se solucion
42.    posicionInica = c
43.    eliminar elmentos de la solucion
44. return tiempoTotal, salidaRuta(ruta)
```

Pseudocódigo matrizPesosMinimos

```
1.  matriz(a,b,ab,ba)
2.  completo = { }
3.    for i = to a.size
4.       completo añadir=a[i]
5.    for j = to a.size
6.       completo añadir=b[j]
```

```
7.  mitad = completo.size/2+1
8.  ultimo = comple.size
9.  contador1=0
10. contador2=mitad+2
11. contador3= 0
12.  for x = to a.size
13.        contador1=x+1
14.        for y = to completo.size
15.          if(x=mitad)
16.            matriz[x][y] = infinito
17.          else if(y=contador1)
18.            matriz[x][y] = completo[y]
19.          else if(y=contador2)
20.            matriz[x][y] = completo[y]+ab[contador3]
21.            contador3 ++
22.          else
23.             matriz[x][y]=infinito
24.        contador2 ++
25. contador1=0
26. contador2=mitad+2
27. contador3= 0
28. for x = from mitad to completo.size
29.        contador1 ++
30.        for y = to completo.size
31.          if(x=ultimo)
32.            matriz[x][y] = infinito
33.          else if(y=contador1)
34.            matriz[x][y] = completo[y]+ba[contador3]
35.            contador3++
36.          else if(y=contador2)
37.            matriz[x][y] = completo[y]
38.            contador3 ++
39.          else
40.             matriz[x][y]=infinito
41.        contador2 ++
42. return matriz
```

Pseudocódigo salidadRuta

```
1.  salidaRuta(Array rutaVertices)
2.    ruta = { }
3.    for i = to rutaVertices.size
4.          if(rutaVertices[ i ] < rutaVertices.size  )
5.             añadir a ruta “a”
6.          else
7.             añadir a ruta “b”
8.  return ruta
```

Relación entre los algoritmos y su implementación

En la carpeta code/voraz se pueden encontrar los archivos que se usaron para la implementación, usando el lenguaje de programación python en su versión 3.9.7. Dentro de esta carpeta se encuentran 5 archivos python que permitieron realizar la implementación y desarrollo de lo requerido para el proyecto.

- voraz.py : Es el archivo principal, en el cual encontramos las funciones que se emplean para ejecutar el algoritmo.

  En La función voraz Inicialmente se encuentra la actividad por la cual se debe comenzar el ensamble, esto lo hacemos calculando el valor mínimo entre los tiempos que tardan la actividad número 1 de a y la actividad número 1 de b, dependiendo del resultado de esta operación nos paramos en el vértice que le corresponde a la actividad escogida, y con esto comparamos las dos posibles siguientes actividades a escoger, escogemos la siguiente actividad calculando el valor mínimo entre los pesos que nos indica nuestra matriz para el vértice en el que estamos parados, lo anterior lo repetimos hasta que se llegue ya sea a la última actividad de a o a la última actividad de b.

  El resultado del algoritmo garantiza una solución válida de acuerdo a las restricciones del problema, pero no siempre esta solución va a ser la más óptima como en el caso del algoritmo dinámico.

  Para la implementación de este algoritmo voraz se utilizó tres funciones auxiliares,

  - Función que nos genera la matriz de pesos mínimos.
  - Función que nos indica el vértice por el cual debemos empezar(línea 1).
  - Función que nos retorna el camino por el cual se hace el ensamblaje.

  Esta última función se hace necesaria ya que el algoritmo voraz retornara un camino escrito en término de los vértices que se escogieron para la solución, por lo anterior debemos indicar a qué línea a o b pertenece cada uno de los vértices de la solución.

- mainVoraz.py: Este archivo se encarga importar las funciones de los archivos voraz.py e inOut.py para poder recibir una entrada y dar una salida en el formato requerido, tiene la función main() que lee un archivo, luego llama al método solve de voraz.py y lo que retorna lo escribe en un nuevo archivo de texto out.txt.
- Los archivos inOut.py, plotting_py.py y test_creation.py funcionan de la misma manera como se explicó anteriormente en dinámica.

Complejidad

la complejidad de la función voraz la podemos deducir de la siguiente manera, las líneas que contienen las operaciones principales para llevar a cabo este algoritmo son la línea 1 y la 26, cada que ejecutemos el algoritmo la línea número 1 se ejecutará una sola vez.

la operación principal la encontramos en la línea 26 la cual tiene un while que se ejecutará mientras el arreglo vérticesCambio no esté vacío, como se puede observar el arreglo vérticesCambio inicialmente va a tener un tamaño de 2n, luego podemos evidenciar en el if que va de la línea 18 hasta la 25 y en el if de que va de la línea 35 hasta la 40, que el tamaño del arreglo verticesCambio se irá disminuyendo de 2 en 2, por lo tanto el while se ejecutará n-1 veces. por lo anterior podemos decir que la complejidad del algoritmo es:

$$1 + \sum_{i=1}^{n-1} 1 = 1 + (n-1)1$$

$$1 + \sum_{i=1}^{n-1} 1 = 1 + (n-1) = n -> O(n)$$

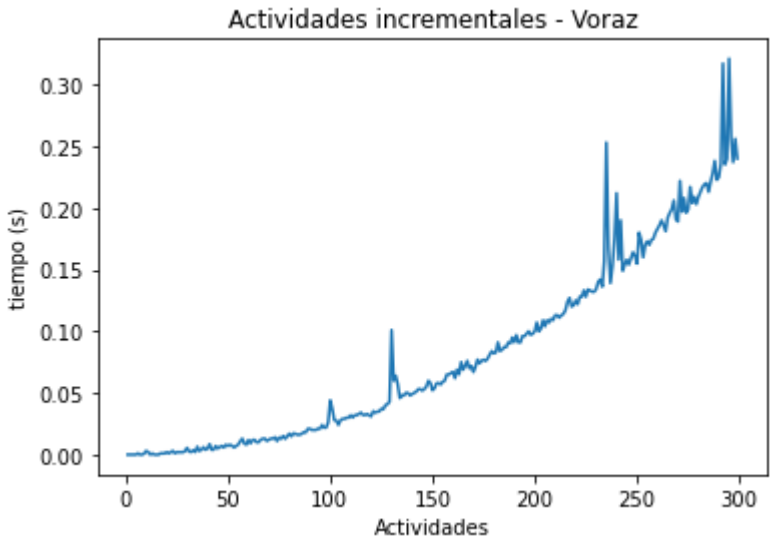
Aunque la función voraz es la función principal para llegar a la solución que buscamos, también hay que tener en cuenta la complejidad de la función matrizPesosMinimos, ya que sin esta no podríamos ejecutar la función voraz a la cual se le pasa como último parámetro una matriz que es el resultado de ejecutar la función matrizPesosMinimos con parámetros a, b, ab, ba. en esta función las operaciones principales las encontramos de la línea 12 a la 24, en donde se ejecuta un primer ciclo anidado, el cual se encarga de llenar nuestra primer parte de la matriz teniendo en cuenta los parámetros a y ab, y de la línea 28 hasta la 41 en donde se encuentra un segundo ciclo que se encarga de llenar la segunda parte de la matriz teniendo en cuenta los parámetros b y ba. estos ciclos iteraran cada uno n*2n por lo que podemos concluir que la complejidad de la función matrizPesosMinimo

$$2(\sum_{i=1}^{n} * \sum_{y=1}^{2n} 1) = 2(n * 2n)$$

$$2(\sum_{i=1}^{n} * \sum_{y=1}^{2n} 1) = 2(n * 2n) \text{->O}(4n^2)$$

Gráficas - Pruebas

Haciendo uso de la función incremental test para el algoritmo voraz se obtuvieron los siguientes resultados:



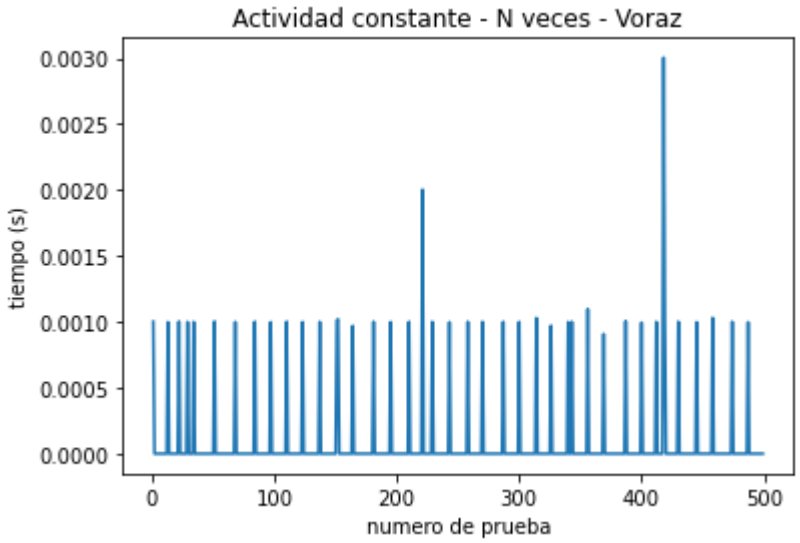
Con una entrada de 300 actividades y un tiempo medio total de `0.08043873589174405`.

En la grafica podemos observar el por qué se debe tener en cuenta la complejidad de la función matrizPesosMinimo, ya que como se explicó anteriormente el programa al ejecutar la función voraz también debe ejecutar la función matrizPesosMinimo, de la cual se dedujo que tenía una complejidad O(4n²), y como se observa en la gráfica el programa al ser ejecutado tiene un comportamiento O(n²) por lo anterior podemos comprobar que la complejidad que obtuvimos teóricamente se asemeja al comportamiento real de la solución.

Pruebas test redundante - Voraz vs dinámica

Test_redundate con voraz y dinámica aplicado en el archivo in2.txt ubicado en el archivo Pruebas/Test_in_files

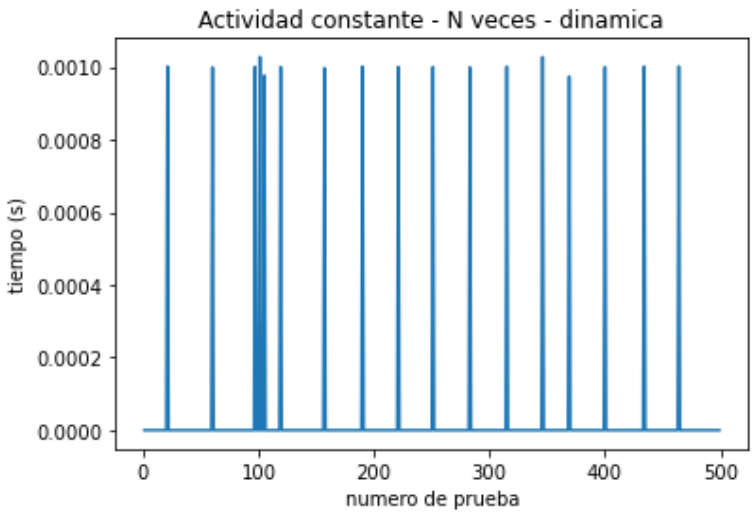
Voraz



Tiempo de ejecución: `8.825453106530444e-05`

Salidas: La salida se ubica en el archivo out2 ubicado en en Pruebas/outVoraz/out2.txt

Dinámica



Tiempo de ejecución: 3.408668992036808e-05

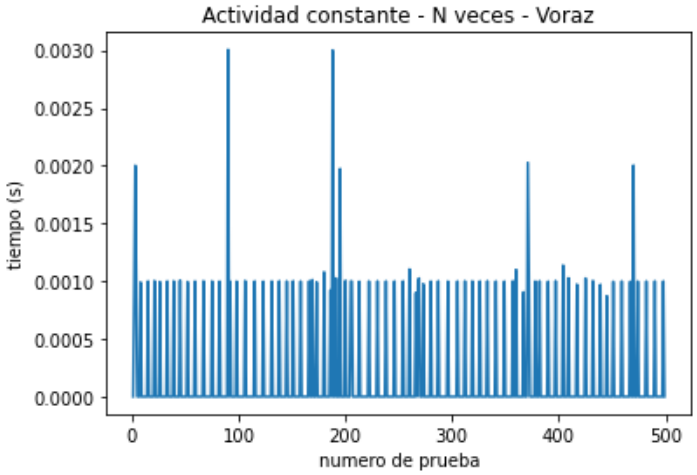
Salidas: La salida se ubica en el archivo out2 ubicado en en Pruebas/outDinamica/out2.txt

Comparación de salidas (voraz vs dinámica)

Las dos salidas dan una solución óptima al problema.

Test_redundate con voraz y dinámica aplicado en el archivo in3.txt ubicado en el archivo Pruebas/Test_in_files

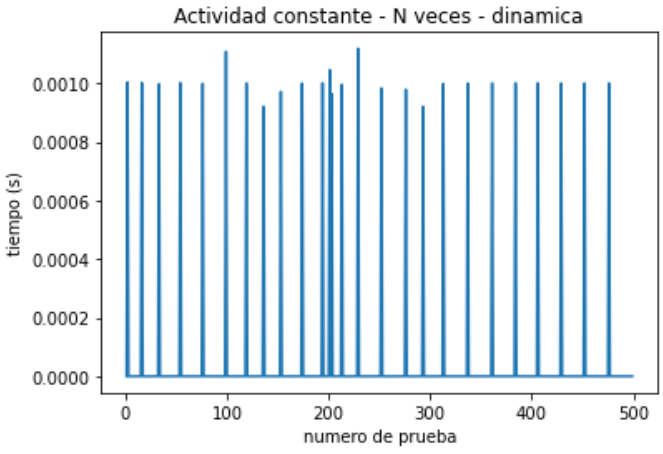
Voraz



Tiempo de ejecución: `0.00018018376612233256`

Salidas: se ubica en el archivo out2 ubicado en en Pruebas/outVoraz/out3.txt

Dinámica



Tiempo de ejecución: 5.2115243518041945e-05

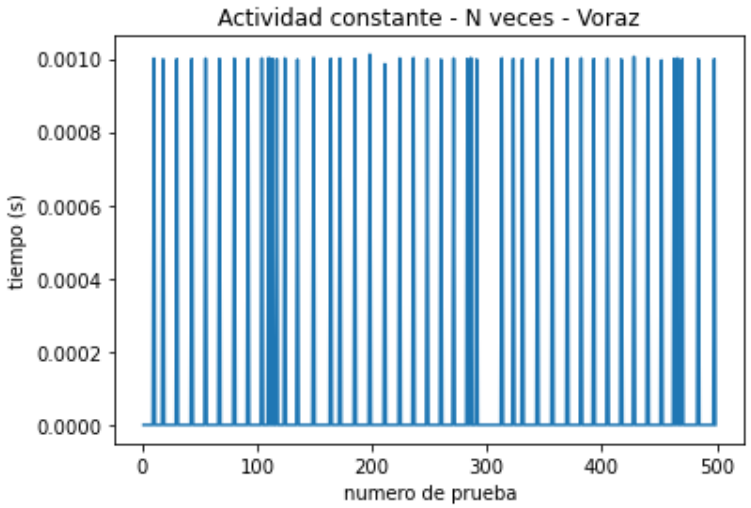
Salidas: La salida se ubica en el archivo out3 ubicado en en Pruebas/outDinamica/out3.txt

Comparación de salidas (voraz vs dinámica)

Para este caso se observa una pequeña diferencia en las salidas ya que el algoritmo voraz y el algoritmo dinámico toman caminos diferentes y esto se ve reflejado en el tiempo total mínimo que para el caso de dinámica es 35 y para el voraz 38. Con esto se puede decir que el algoritmo de programación dinámica da la solución óptima pero el algoritmo voraz no, a pesar de que se acerca a la solución.

Test_redundate con voraz y dinámica aplicado en el archivo in10.txt ubicado en el archivo Pruebas/Test_in_files

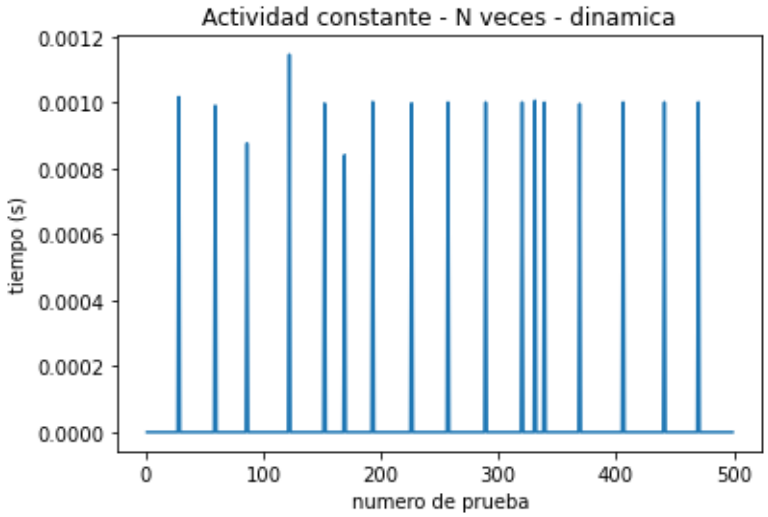
Voraz



Tiempo de ejecución: 9.220396588465016e-05

Salidas: se ubica en el archivo out10 ubicado en en Pruebas/outVoraz/out10.txt

Dinámica



Tiempo de ejecución: 3.380144765238485e-05

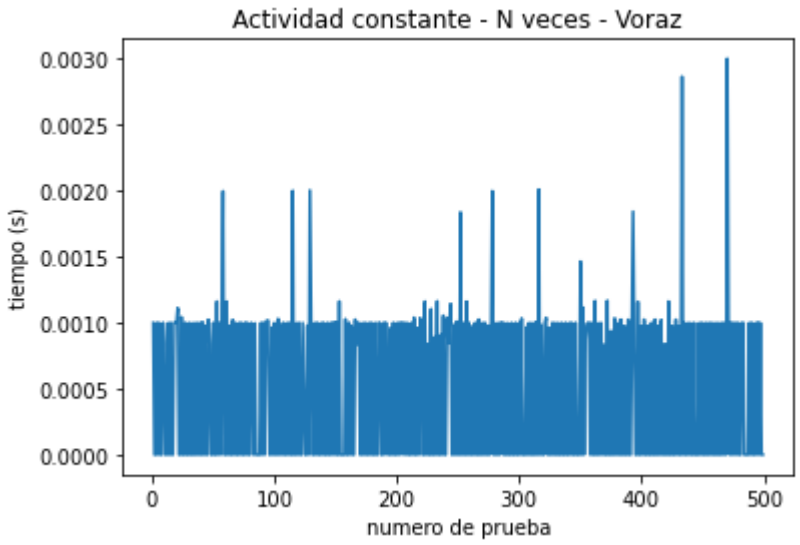
Salidas: La salida se ubica en el archivo out10 ubicado en en Pruebas/outDinamica/out10.txt

Comparación de salidas (voraz vs dinámica)

Este caso es un poco peculiar porque los dos algoritmos dan diferentes soluciones óptimas al problema ya que minimizan el tiempo, para ambos el tiempo mínimo es de 21 pero toman diferentes caminos.

Test_redundate con voraz y dinámica aplicado en el archivo in12.txt ubicado en el archivo Pruebas/Test_in_files

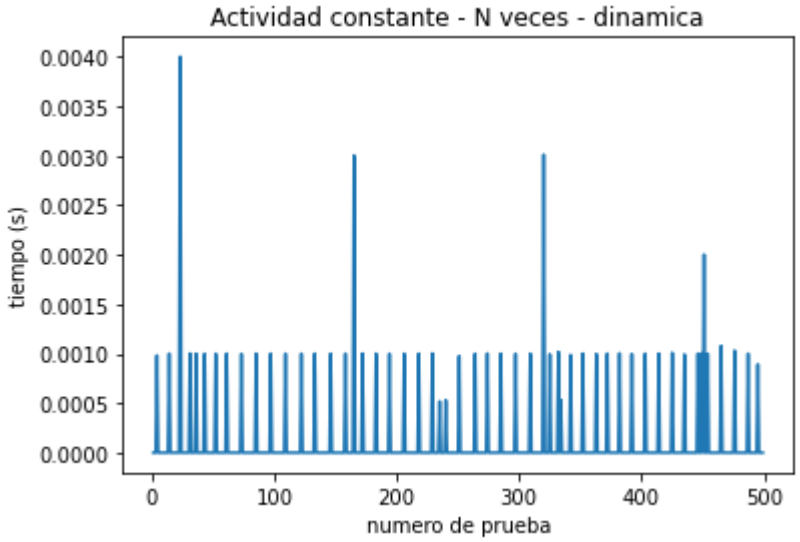
Voraz



Tiempo de ejecución: `0.0004397365516555572`

Salidas: se ubica en el archivo out.2 ubicado en en Pruebas/outVoraz/out12.txt

Dinámica



Tiempo de ejecución: 0.0001193526273738884

Salidas: La salida se ubica en el archivo out12 ubicado en en Pruebas/outDinamica/out12.txt

Comparación de salidas (voraz vs dinámica)

Para este caso se observa una pequeña diferencia en las salidas ya que el algoritmo voraz y el algoritmo dinámico toman caminos diferentes y esto se ve reflejado en el tiempo total mínimo que para el caso de dinámica es 368 y para el voraz 399. Con esto se puede decir que el algoritmo de programación dinámica da la solución óptima pero el algoritmo voraz no, a pesar de que se acerca a la solución. Con una diferencia solo de una sola actividad que en el algoritmo dinámico se realiza la actividad número 7 en la línea ‘a’ y en el voraz en la línea ‘b’.

Comparación de tiempos para los tests in2, in3, in10 y in12

```
In [1]: 8.825453106530444e-05 < 3.408668992036808e-05
Out[1]: False

In [2]: 0.00018018376612233256 < 5.2115243518041945e-05
Out[2]: False

In [3]: 9.220396588465016e-05 < 3.380144765238485e-05
Out[3]: False

In [4]: 0.0004397365516555572 < 0.0001193526273738884
Out[4]: False
```

Se evidencia al comparar los tiempos de las pruebas tanto en voraz como en dinámica, que el algoritmo dinámico es más eficiente en su tiempo de ejecución, esto se da por lo anteriormente explicado en la página 20, el algoritmo voraz al tener que hacer uso de la función matrizPesosMinimos se está viendo afectado al momento de hacer su ejecución.

Una posible solución a lo anterior es realizar otra función que se encargue de llenar la matriz de pesos mínimos, con una complejidad menor o igual a la complejidad del algoritmo voraz.

Repositorio de Github con implementación en python

https://github.com/Odzen/ProyectoFinal_FADA