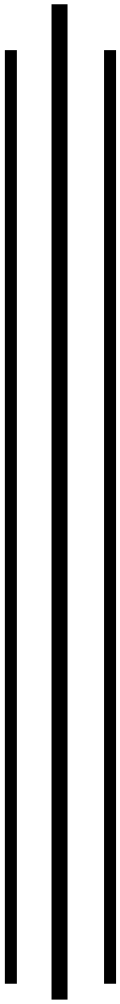
INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN



CURSO PROPEDÉUTICO
PROGRAMACIÓN Y ALGORITMIA

Proyecto final

Caracterización experimental de algoritmos para planificación de intervalos.

PRESENTA:

Tirado Torres Ricardo Aldair

06 de diciembre del 2022

1. Descripción del problema

La maximización de pesos de un conjunto de intervalos o planificación de intervalos por pesos es un problema muy famoso en el campo de las ciencias de la computación en el que se proporciona una lista de diferentes trabajos, con la hora de inicio, la hora de finalización y la ganancia de ese trabajo (peso). La solución del problema se da encontrando un subconjunto de trabajos, donde la ganancia sea máxima y ningún trabajo tenga conflictos con otros en el tema de tiempos en conflicto. Para ello se pueden utilizar diversas estrategias de programación, dos de las más conocidas son el algoritmo *Greedy*, cuyo enfoque se da más hacia intervalos sin pesos, y los algoritmos de programación dinámica que aprovechan el paradigma de diseño algorítmico “divide y vencerás”. Ahora bien, dentro de esta última estrategia hay dos vertientes que se planean considerar para este proyecto, que son la programación dinámica sin memoización y la *Bottom up*.

El problema que se desea resolver es la caracterización de estos 3 tipos de soluciones para el problema de la planificación de intervalos, por lo que se necesita compararlos por medio de una gráfica que muestre el peso de las soluciones y el tiempo que tardan en hallar dichos valores. También se requiere que la planificación de intervalos sea una instancia completamente aleatoria, para reducir o eliminar, un posible sesgo por usar valores arbitrarios. Otra necesidad que se debe satisfacer es que los números obtenidos de forma aleatoria tenga una distribución uniforme (0,1), para que el rango de valores de inicio, fin y peso de los intervalos sea igualmente probable.

Finalmente, se desea crear el algoritmo utilizando el lenguaje de programación Python, pero utilizando la menor cantidad de librerías de programación, por lo que solo se podrán emplear para graficar los valores obtenidos en las soluciones. De ahí en más, los algoritmos necesarios para la generación de intervalos y sus soluciones deben ser hechos sin el uso de trabajo de terceros (módulos).

2. Justificación de la propuesta

En los tiempos actuales, la planificación resulta vital para el desarrollo de metas personales de un individuo, pero su alcance va más allá, debido a que las empresas al establecer objetivos, formulan estrategias para alcanzarlos, pero para ello se requiere de una organización, coordinación y control de actividades, y es aquí en donde entra la planificación de acciones en intervalos de tiempos como función administrativa.

Ahora bien, en un inicio los trabajos que se necesitan efectuar pueden estar sobre encimados en el tiempo con otros, por lo que se tiene que decidir cuales realizar, incluso si requiere descartar a otras. Esta toma de decisiones no solo debe considerar que las actividades escogidas sean compatibles, sino que se deben priorizar las actividades que en conjunto tengan el mayor grado de importancia posible. Ante un número bajo de trabajos la elección puede ser sencilla, no obstante, conforme incrementa el número de actividades, aumentará también el tiempo en que se desarrolla la planificación. Es por ello que se necesita un método de selección de actividades que no solo arroje el conjunto de trabajos de mayor valor, sino que la planificación debe realizarse lo más rápido posible, puesto que, en un mercado laboral cada vez más competitivo, la optimización de tiempos asegura una mayor productividad y rendimiento.

Por lo tanto, se necesita ejemplificar como es que, no siempre, el algoritmo más rápido o el que entrega el resultado óptimo es la mejor opción, debido a que se necesitan desarrollar estrategias o en su momento, sacrificar elementos como el espacio para hallar la mejor solución

3. Estado del arte

La investigación sobre la planificación de intervalos se ha llevado a cabo durante mucho tiempo y no únicamente aplicando estrategias *Greedy* y de programación dinámica, ya que existen casos en los que se utilizan otros algoritmos, pero aplicados a la vida real, como por ejemplo la propuesta realizada por Yang *et al*, que desarrolló algoritmos de planeación fuera de línea y planeación en línea para sistemas ciberfísicos (CPS) marítimos, para abordar un problema de entrega de videos de vigilancia, que los buques generan desde el puerto de origen hasta el puerto de destino. Se supone que, durante la navegación, las estaciones de información en alta mar podrían cargar los videos de vigilancia. Los videos se dividen en paquetes, que podrían entregarse con éxito antes de sus tiempos límite. Para ello se actúa a través del problema de la máquina de trabajo matemático. Y cada trabajo se revela en un tiempo de lanzamiento, una fecha límite, un tiempo de procesamiento y un peso. Para maximizar el peso de los trabajos, propusieron tres algoritmos: un algoritmo fuera de línea, un algoritmo de admisión en línea sin tiempos de procesamiento acotados, así como un algoritmo de capacidad exponencial con tiempos de procesamiento acotados. Los resultados de la simulación verificaron el desempeño de los mismos. Se compararon los algoritmos propuestos con algunos otros algoritmos de programación clásicos, es decir, fecha límite (el trabajo con el tiempo límite más temprano se programa primero), *First-input-first-output* (FIFO) (el trabajo con la hora de lanzamiento más temprana se programa primero), Peso (el trabajo con el mayor peso se programa primero). El resultado fue que, con un mayor tiempo entre llegadas, la cantidad de paquetes superpuestos disminuyó y el rendimiento general aumentó. El algoritmo de admisión y el algoritmo de capacidad exponencial fueron mejores que los otros algoritmos. [1]

Otro ejemplo es el que realizó Saito y Chiba, quienes después de que se demostrará que maximizar el peso total de los trabajos *just in time* en condiciones de múltiples ranuras es NP-hard, consideraron un algoritmo heurístico. Primero, calcularon un programa que minimiza el número de intervalos de tiempo. Después, crearon un nuevo conjunto de trabajos al fusionar los trabajos existentes en la planeación. Luego, calcularon un flujo de costo mínimo para la red construida a partir de este nuevo conjunto de trabajos. Y finalmente, obtuvieron un horario factible del flujo. Además, implementaron el algoritmo heurístico y consideraron sus características y rendimiento a partir de la experimentación computacional. [2]

Finalmente, y retomando el concepto de trabajos o actividades *just in time*, se tiene el caso de Kawamata y Sung, quienes lidiaron con el problema de maximizar el peso de trabajos pseudo *just in time* en el modelo de una sola máquina. En su formulación, cada trabajo está asociado con una ventana de tiempo en lugar de una fecha de vencimiento, y la duración de todas esas ventanas de tiempo está limitada arriba por un cierto valor dado por adelantado. Un trabajo se llama pseudo *just in time* si se completa dentro de su ventana de tiempo. Por lo que terminaron demostrando que el problema es resoluble en tiempo polinomial y, además, mostraron algunos resultados sobre la complejidad computacional para configuraciones más generales. [3]

Como se mostró en los ejemplos, siempre existirán algunas nuevas propuestas para optimizar valores de pesos y mejorar la complejidad de tiempo. Por lo tanto, la caracterización del problema de planificación de intervalos resulta importante para entender de manera sencilla como es que un enfoque diferente puede traer resultados más óptimos a costes de tiempo accesibles.

4. Solución de la propuesta

Actualmente ya se cuentan con varios algoritmos que solucionan el famoso problema de la planificación de intervalos (*Interval Scheduling Maximization problem*), no obstante, algunos de estos entregan una solución aproximada a la óptima y otros, conforme aumenta la cantidad de actividades, tienen una complejidad en el tiempo alta, que los hace inviables a partir de un cierto número de trabajos a procesar. Es por ello que en este trabajo se desea caracterizar los resultados de optimización y tiempo de ejecución de tres estrategias particulares en las ciencias de computación: El algoritmo voraz (*Greedy*), la programación dinámica por fuerza bruta (sin memoización) y la programación dinámica del tipo "*Bottom up*".

Para ello se implementará un generador de instancias aleatorias del problema de planificación de intervalos con pesos usando la distribución uniforme. De acuerdo a los resultados obtenidos por cada algoritmo, se visualizará en una gráfica el valor promedio, mínimo y máximo del peso de la solución encontrada en 10 instancias aleatorias en función de 10, 50, 100 y 200 intervalos. En otra gráfica se observará el valor promedio, mínimo y máximo del tiempo de ejecución de las mismas 10 instancias aleatorias y los mismos tamaños de entrada.

La solución del problema se implementará por medio de un algoritmo general, el cual sigue de los siguientes pasos:

- Generar una instancia de intervalos cuyos atributos (inicio, longitud y peso) sean aleatorios y con una distributividad uniforme (Se utilizará un Generador Congruencial Lineal para obtener los valores).
- Ordenar los intervalos de la instancia de acuerdo al tiempo de finalización más temprano.
- Generar una lista que almacene los trabajos compatibles de cada intervalo.
- Ejecutar la solución *Greedy* y guardar el resultado.
- Ejecutar la solución de programación dinámica (sin memoización) y guardar el resultado.
- Ejecutar la solución *Bottom up* y guardar el resultado.
- Almacenar los valores de tiempo de ejecución de cada una de las soluciones.
- Calcular valores promedios, máximos y mínimos de cada solución.
- Graficar los valores obtenidos.

5. Implementación con el código fuente

5.1 Generador de números aleatorios dentro de un rango de valores

```
 9  def NumeroRandomLCG(min,max):
10      global rand
11      max += 1
12      a = 7**5
13      m = (2**31)-1
14      rand = (a*rand) % m
15      prob = rand/m
16      return int(((max-min)*(prob)) + min)
...
454     rand = int(time.time())
```

Antes de comenzar la explicación del código implementado, es muy importante entender el funcionamiento del algoritmo de números pseudoaleatorios: “NumeroRandomLCG”. Esta función recibe dos parámetros, “min” y “max”, que tienen como propósito delimitar los números calculados al rango que hay entre estos dos. Esta función emplea un generador lineal congruencial (LCG, por sus siglas en inglés), que es un método que permite obtener una secuencia de números pseudoaleatorios calculados con una función lineal definida a trozos discontinua. [4]

Para comenzar a calcular los números pseudoaleatorios se requiere de una semilla, la cual, para este trabajo, se trata de una medición de tiempo (variable “rand”). El número obtenido se divide entre el valor de *module* (variable “m”) para obtener un valor (0, 1) uniforme (variable “prob”). Por último, se multiplica a “prob” por la diferencia de “max” y “min” y se le agrega el valor de este último, para obtener un número distribuido uniformemente entre los 2 parámetros de entrada.

5.2 Generador de instancias

```
18  def GeneradorDeInstancias(n):
19      global LongitudMin
20      global LongitudMax
21      global PesoMin
22      global PesoMax
23      global TiempoMax
24      planificacion = []
25      for i in range(n):
26          start = NumeroRandomLCG(0,TiempoMax)
27          end = start + NumeroRandomLCG(LongitudMin,LongitudMax)
28          weight = NumeroRandomLCG(PesoMin,PesoMax)
29          planificacion.append((start,end,weight))
30      return planificacion
...
217             G = GeneradorDeInstancias(j)
...
449     LongitudMin = 30
450     LongitudMax = 50
451     PesoMin = 1
452     PesoMax = 50
453     TiempoMax = 100
```

Para generar una instancia de n intervalos se utiliza la función “GeneradorDeInstancias”, que tiene como parámetro de entrada a “n”, el cual define el número de intervalos que almacenará la lista “Planificacion”. Las cinco variables globales se utilizan para llamar a la función de los números pseudoaleatorios y delimitar el rango de valores obtenidos para el inicio, fin y peso de cada intervalo. Una vez termina el bucle, la variable global “G” almacena en una lista los valores de cada intervalo.

5.3 Ordenamiento por mezcla

```
32 def Mezclar(arr, l, m, r):
33     n1 = m - l + 1
34     n2 = r - m
35     L = [0] * (n1)
36     R = [0] * (n2)
37     for i in range(0, n1):
38         L[i] = arr[l + i]
39     for j in range(0, n2):
40         R[j] = arr[m + 1 + j]
41     i = 0
42     j = 0
43     k = l
44     while i < n1 and j < n2:
45         if L[i][1] <= R[j][1]:
46             arr[k] = L[i]
47             i += 1
48         else:
49             arr[k] = R[j]
50             j += 1
51         k += 1
52     while i < n1:
53         arr[k] = L[i]
54         i += 1
55         k += 1
56     while j < n2:
57         arr[k] = R[j]
58         j += 1
59         k += 1
60
61 def OrdenarPorMezcla(arr, l, r):
62     if l < r:
63         m = l+(r-l)//2
64         OrdenarPorMezcla(arr, l, m)
65         OrdenarPorMezcla(arr, m+1, r)
66         Mezclar(arr, l, m, r)
...
219             OrdenarPorMezcla(G, 0, len(G)-1)
```

Una vez se tienen la lista de intervalos aleatorios, se necesitan ordenar a los elementos de la instancia de acuerdo al Lema: “Al menos un máximo conjunto de intervalos libre de conflictos incluye el intervalo que termina primero” [5]. Para ello se ocupa el *Merge sort* u ordenamiento por mezcla, que es un algoritmo de divide y vencerás basado en la idea de dividir una lista en varias sublistas hasta que cada sublista consista en un solo elemento y fusionar esas sublistas de una manera que resulte en una lista ordenada. La función “OrdenarPorMezcla” recibe de argumentos al arreglo (lista, en este caso), su primer índice y el último, los cuales son

empleados para dividir los índices por medio de recursividad. Una vez se tienen los índices de los subarreglos del menor tamaño posible, se llama a la función “Mezclar” que genera los subarreglos (“L” y “R”) y los compara de acuerdo a su tiempo de finalización (“L[i][1]” y “R[j][1]”) de menor a mayor y desde los subarreglos más pequeños, hasta regresar a la lista original, pero con sus intervalos ordenados.

5.4 Trabajos compatibles

```
139 def TrabajosCompatibles():
140     global G
141     P = [0]*len(G)
142     for i in range(len(G)):
143         for j in range(i-1,-1,-1):
144             if G[i][0] >= G[j][1]:
145                 P[i] = j+1
146                 break
147     return P
```

Con la lista de intervalos ordenada como se indica en el Lema, se podría obtener la solución por medio del Algoritmo *Greedy*, pero antes de ello se procede a generar una nueva lista que almacenará la cantidad de actividades compatibles de cada intervalo usando “TrabajosCompatibles”. La función crea una variable tipo lista con elementos de valor 0 debido a que en un inicio se supone que ningún trabajo es compatible, ahora bien, el primer bucle sirve para indicar el intervalo al que se le identificarán las actividades compatibles y el segundo bucle compara a cada actividad con el intervalo, sin embargo, en este proyecto se optó por una cuenta descendente, lo que quiere decir que, el primer intervalo no tendrá actividades compatibles por no tener actividades después de él, pero el último si, por considerar a todas las actividades que no tengan conflicto con su valor de inicio. También cabe señalar que lo que se guarda en “P” no son los índices, sino el número o cantidad de actividades compatibles, lo cual tiene relación con la siguiente fórmula:

$$\text{índice} + 1 = \text{actividad compatible}$$

5.5 Algoritmo Greedy

```
149 def AlgoritmoVoraz():
150     global G
151     SolucionVoraz = []
152     Peso = 0
153     for j in G:
154         if len(SolucionVoraz) == 0:
155             SolucionVoraz.append(j)
156         if SolucionVoraz[-1][1] <= j[0]:
157             SolucionVoraz.append(j)
158     for elemento in SolucionVoraz:
159         Peso += elemento[2]
160     return Peso
...
223             S1.append(AlgoritmoVoraz())
```

Con la instancia de intervalos aleatorios “G”, creada y ordenada, se procede a hallar la primera solución, por el método de voracidad o *greedy*. Esta estrategia se utiliza en problemas de

optimización. El algoritmo hace la elección óptima en cada paso mientras intenta encontrar la forma óptima general de resolver todo el problema [6]. Para el caso del problema de selección de actividades, la función “AlgoritmoVoraz” selecciona a la primera actividad de la lista “G” y la agrega a la lista “SolucionVoraz”, a partir de ahí, si el valor de inicio de la actividad seleccionada actualmente es mayor o igual que la hora de finalización de la actividad seleccionada anteriormente (última actividad añadida), se agrega a la lista “SolucionVoraz”. Una vez se tiene el subconjunto de intervalos seleccionados por el algoritmo, se suman los valores de pesos y se retorna como la solución.

5.6 Algoritmo de programación dinámica (sin memoización)

```
162 def PDSinMemo(j):
163     global G
164     global P
165     if j == -1:
166         return 0
167     else:
168         return max(G[j][2] + PDSinMemo((P[j])-1), PDSinMemo(j-1))
...
225             S2.append(PDSinMemo(len(G)-1))
```

Para la segunda solución se emplea el concepto de programación dinámica, que es una técnica estándar en el diseño de algoritmos en la que se construye una solución óptima para un problema a partir de soluciones óptimas para una serie de subproblemas, normalmente almacenados en una tabla o matriz multidimensional [7]. Para la segunda solución de este proyecto, se utilizará la forma recursiva de la programación dinámica, pero sin almacenar la solución de cada subproblema, en otras palabras, sin memoización. Esto se realiza con la función “PDSinMemo”, que recibe el índice del último elemento de la lista de intervalos y por medio de recursividad se obtiene el valor de peso máximo de la comparación entre el intervalo actual, más la llamada de la función “PDSinMemo” para el siguiente intervalo no conflictivo, y la llamada de la misma función para el intervalo siguiente al actual (que puede ser conflictivo o no). De esta forma se va generando un árbol binario para cada intervalo junto con la suma de los intervalos no conflictivos de este.

Una vez que se llega a la última ramificación (P[j] regresa 0 actividades, o sea, un índice -1), se retorna el valor de 0 puesto que ya no hay ningún intervalo no conflictivo, por lo que ahora el árbol va desde la última ramificación hasta el origen, retornando como solución el valor máximo de pesos del conjunto óptimo de intervalos no conflictivos.

5.7 Algoritmo de programación dinámica (*Bottom up*)

```
170 def PDBottomUp():
171     global G
172     global P
173     M = [0]*(len(G)+1)
174     M[1] = G[0][2]
175     for j in range(1,len(G)+1):
176         M[j] = max(G[j-1][2] + M[P[j-1]], M[j-1])
177     return M[-1]
...
227             S3.append(PDBottomUp())
```

Finalmente, para la última solución se utilizará el concepto de programación dinámica, pero desde un enfoque iterativo, almacenando la solución óptima de los subproblemas, para evitar realizar acciones repetitivas que aumentan la complejidad de tiempo. Como se observa, en la función “PDBottomUp” no se tiene un parámetro de entrada, se vuelve a manipular la lista de intervalos no conflictivos y se genera una lista en donde se guardarán las soluciones de los subproblemas más sencillos, para que conforme continua el bucle, los problemas más complejos se resuelvan con los resultados almacenados. Una vez que se termine el bucle, la solución retornada por esta estrategia es el último dato almacenado en la lista “M”.

5.8 Tiempos de ejecución de cada solución

```
222             Inicio1 = time.time()
223             S1.append(AlgoritmoVoraz())
224             Fin1 = time.time()
225             S2.append(PDSinMemo(len(G)-1))
226             Fin2 = time.time()
227             S3.append(PDBottomUp())
228             Fin3 = time.time()
...
232             T1.append(Fin1-Inicio1)
233             T2.append(Fin2-Fin1)
234             T3.append(Fin3-Fin2)
```

Al mismo tiempo que se obtienen las soluciones de los pesos, se calcula el tiempo, empleando la función “time()” de la librería homónima, para almacenar en 4 variables, los tiempos de inicio y termino de cada algoritmo. Para conocer el tiempo que tardó cada solución, se hace la sustracción de una variable con otra, según corresponda.

5.9 Calcular de valores promedios, máximos y mínimos

```
215     for j in Entradas:
216         for i in range(Instancias):
217             G = GeneradorDeInstancias(j)
218             Intervalos.append(G.copy())
219             OrdenarPorMezcla(G, 0, len(G)-1)
220             IntervalosOrdenados.append(G)
221             P = TrabajosCompatibles()
222             Inicio1 = time.time()
223             S1.append(AlgoritmoVoraz())
224             Fin1 = time.time()
225             S2.append(PDSinMemo(len(G)-1))
226             Fin2 = time.time()
227             S3.append(PDBottomUp())
228             Fin3 = time.time()
229             ValoresPromediosS1[k] += S1[-1]
230             ValoresPromediosS2[k] += S2[-1]
231             ValoresPromediosS3[k] += S3[-1]
232             T1.append(Fin1-Inicio1)
233             T2.append(Fin2-Fin1)
234             T3.append(Fin3-Fin2)
235             TiemposPromediosS1[k] += T1[-1]
236             TiemposPromediosS2[k] += T2[-1]
237             TiemposPromediosS3[k] += T3[-1]
238
```

```
239         ValoresPromediosS1[k] = float(ValoresPromediosS1[k])/10
240         ValoresPromediosS2[k] = float(ValoresPromediosS2[k])/10
241         ValoresPromediosS3[k] = float(ValoresPromediosS3[k])/10
242         ValoresMaximosS1.append(max(S1))
243         ValoresMaximosS2.append(max(S2))
244         ValoresMaximosS3.append(max(S3))
245         ValoresMinimosS1.append(min(S1))
246         ValoresMinimosS2.append(min(S2))
247         ValoresMinimosS3.append(min(S3))
248
249         TiemposPromediosS1[k] = float(TiemposPromediosS1[k])/10
250         TiemposPromediosS2[k] = float(TiemposPromediosS2[k])/10
251         TiemposPromediosS3[k] = float(TiemposPromediosS3[k])/10
252         TiemposMaximosS1.append(max(T1))
253         TiemposMaximosS2.append(max(T2))
254         TiemposMaximosS3.append(max(T3))
255         TiemposMinimosS1.append(min(T1))
256         TiemposMinimosS2.append(min(T2))
257         TiemposMinimosS3.append(min(T3))
258         k += 1
259         S1 = []
260         S2 = []
261         S3 = []
262         T1 = []
263         T2 = []
264         T3 = []
```

Todo lo visto hasta el momento, ha sido para una sola instancia, no obstante, en la propuesta de solución se plantea la generación de 10 instancias de 4 entradas (10, 50 100 y 200 intervalos), por lo que se debe ejecutar 2 bucles:

- Primer bucle: Se ejecutan las 10 instancias, almacenando en cada ciclo las soluciones en las listas “S1”, “S2”, “S3”, “T1”, “T2” y “T3”.
- Segundo bucle: Se ejecuta el cálculo de los promedios, máximos y mínimos con las funciones “max()”, “min()” y la fórmula para hallar el promedio:

$$Promedio = \frac{\sum_0^9 ValoresPromedioS}{instancias}$$

  Una vez calculados, se guardan estos valores finales en nuevas variables, debido a que las variables usadas serán reutilizadas para la ejecución de nuevas instancias, pero ahora con otro número de intervalos.

5.10 Graficado de los pesos y tiempos de ejecución.

```
79 def
GraficarSoluciones(Entradas,p_S1,p_S2,p_S3,max_S1,max_S2,max_S3,min_S1,min_S2,mi
n_S3,Titulo,EjeY):
80      if Titulo == "Pesos de la solución":
81          FigV = plt.figure("Soluciones de pesos promedios")
82      else:
83          FigV = plt.figure("Tiempos de ejecución promedios")
85      ax = FigV.add_subplot(1,1,1)
86      x = np.array(Entradas)
```

```
87      y1 = np.array(p_S1)
88      y2 = np.array(p_S2)
89      y3 = np.array(p_S3)
90      XY1Spline = make_interp_spline(x, y1)
91      XY2Spline = make_interp_spline(x, y2)
92      XY3Spline = make_interp_spline(x, y3)
93      X = np.linspace(x.min(), x.max(), 500)
94      Y1 = XY1Spline(X)
95      Y2 = XY2Spline(X)
96      Y3 = XY3Spline(X)
98      plt.plot(X, Y1, color="green")
99      plt.plot(X, Y2, color="red")
100     plt.plot(X, Y3, color="blue")
102     plt.scatter(Entradas, p_S1, color="green")
103     plt.scatter(Entradas, max_S1, color="green")
104     plt.scatter(Entradas, min_S1, color="green")
105     plt.scatter(Entradas, p_S2, color="red")
106     plt.scatter(Entradas, max_S2, color="red")
107     plt.scatter(Entradas, min_S2, color="red")
108     plt.scatter(Entradas, p_S3, color="blue")
109     plt.scatter(Entradas, max_S3, color="blue")
110     plt.scatter(Entradas, min_S3, color="blue")
112     for i in range(len(Entradas)):
113         if Titulo == "Pesos de la solución":
114             plt.text(Entradas[i]-1, p_S1[i]-6, p_S1[i], color="green",
fontsize='x-small')
115             plt.text(Entradas[i]-1, max_S1[i]-6, max_S1[i], color="green",
fontsize='x-small')
116             plt.text(Entradas[i]-1, min_S1[i]-6, min_S1[i], color="green",
fontsize='x-small')
117             plt.text(Entradas[i]-1, p_S2[i]-6, p_S2[i], color="red",
fontsize='x-small')
118             plt.text(Entradas[i]-1, max_S2[i]-6, max_S2[i], color="red",
fontsize='x-small')
119             plt.text(Entradas[i]-1, min_S2[i]-6, min_S2[i], color="red",
fontsize='x-small')
120             plt.text(Entradas[i]-1, p_S3[i]+4, p_S3[i], color="blue",
fontsize='x-small')
121             plt.text(Entradas[i]-1, max_S3[i]+4, max_S3[i], color="blue",
fontsize='x-small')
122             plt.text(Entradas[i]-1, min_S3[i]+4, min_S3[i], color="blue",
fontsize='x-small')
123         else:
124             plt.text(Entradas[i]-1, p_S1[i]-0.05, p_S1[i], color="green",
fontsize='x-small')
125             plt.text(Entradas[i]-1, max_S1[i]-0.05, max_S1[i], color="green",
fontsize='x-small')
126             plt.text(Entradas[i]-1, min_S1[i]-0.05, min_S1[i], color="green",
fontsize='x-small')
127             plt.text(Entradas[i]-6, p_S2[i], str(p_S2[i])[0:6], color="red",
fontsize='x-small')
128             plt.text(Entradas[i]-6, max_S2[i], str(max_S2[i])[0:6],
color="red", fontsize='x-small')
129             plt.text(Entradas[i]-6, min_S2[i], str(min_S2[i])[0:6],
color="red", fontsize='x-small')
130             plt.text(Entradas[i]-1, p_S3[i]+0.05, p_S3[i], color="blue",
fontsize='x-small')
```

```
131             plt.text(Entradas[i]-1, max_S3[i]+0.05, max_S3[i], color="blue",
fontsize='x-small')
132             plt.text(Entradas[i]-1, min_S3[i]+0.05, min_S3[i], color="blue",
fontsize='x-small')
133     plt.title(Titulo)
134     plt.ylabel(EjeY)
135     plt.xlabel("Número de intervalos")
136     plt.legend(["Greedy","PD (Sin memoización)","PD (Bottom up)"], loc="upper
left")
137     plt.show()
...
266     ParametrosGraficaV =
[Entradas,ValoresPromediosS1,ValoresPromediosS2,ValoresPromediosS3,ValoresMaximo
sS1,ValoresMaximosS2,ValoresMaximosS3,ValoresMinimosS1,ValoresMinimosS2,ValoresM
inimosS3,"Pesos de la solución","Peso"]
269     ParametrosGraficaT =
[Entradas,TiemposPromediosS1,TiemposPromediosS2,TiemposPromediosS3,TiemposMaximo
sS1,TiemposMaximosS2,TiemposMaximosS3,TiemposMinimosS1,TiemposMinimosS2,TiemposM
inimosS3,"Tiempo de ejecución","Tiempo (s)"]
```

Para el graficado de las soluciones de pesos y de los tiempos de ejecución, se utiliza la librería “matplotlib.pyplot”, específicamente las funciones integradas en este módulo:

- Plot(): Se usa para graficar líneas, que en el proyecto representan los valores promedios de las soluciones. Toma de argumentos, principalmente, un arreglo de valores para el eje “x” y otro arreglo para el eje “y”.
- Scatter(): Se emplea para graficar puntos, que en el proyecto representan los valores mínimos y máximos de las soluciones. Toma de argumento, un valor numérico para la posición el eje “x” y un valor para la posición del “y”.
- Text(): Se utiliza para escribir texto, que en el proyecto son los valores numéricos de las solución promedio, mínimos y máximos. Toma de argumento, un valor numérico para la posición del eje “x” y un valor para la posición del “y”.

Además de las funciones mencionadas, se maneja la función “make_interp_spline” de la librería “scipy” puesto que se requiere visualizar curvas más “suaves”, por lo que se utilizó la función para calcular y graficar los coeficientes de interpolación *B-spline*, un tipo de curva *Spline*.

5.11 Interfaz para la ejecución del código implementado

Finalmente, para generar gráficas de las soluciones, pero con la posibilidad de cambiar parámetros de los intervalos, como lo son la longitud mínima y máxima, el peso mínimo y máximo y el tiempo máximo de inicialización de un intervalo, se utiliza una interfaz sencilla empleando la librería de “Tkinter”. En una ventana emergente, se cuentan con botones, para observar los distintos tipos de gráficos y con cuadros de texto, para modificar los valores predeterminados.

El código implementado correspondiente a la interfaz no se analizará a fondo debido a que resulta muy repetitivo el uso de botones, etiquetas de texto y cuadros de textos, cuyos parámetros son la ubicación, el tamaño y el tipo de fuente para el texto.

6. Resultados y Conclusiones

El resultado obtenido se puede ver en la Figura 1, en donde se tienen los diferentes botones que ejecutarán una acción de acuerdo a la etiqueta:

- “Generar nuevas instancias”: Realiza la generación completa de todas las instancias de cada intervalo de entrada, ejecuta los 3 algoritmos, calcula los valores promedios, máximos y mínimos y guarda todos los datos de soluciones.
- “Mostrar gráfica de pesos”: Se genera la gráfica de pesos óptimos de las 3 soluciones en base a los datos almacenados (Ver Figura 2).
- “Mostrar gráfica de tiempos de ejecución”: Se genera la gráfica de tiempos de ejecución de las 3 soluciones en base a los datos almacenados (Ver Figura 2).
- “Mostrar intervalos”: Se puede observar de forma gráfica, una de las instancias de intervalos. Para seleccionar que número de instancias y cuantos intervalos se emplean los cuadros de texto de la parte inferior.
- “Mostrar intervalos ordenados”: Mismo funcionamiento del botón anterior, solo que ahora se visualiza la instancia después de ser ordenada.
- Para los cuadros de texto, únicamente se escribe el valor número entero y se guarda presionando el botón “OK” correspondiente. Cada cuadro de texto cuenta con solución de captura y gestión de errores ante datos no validos escritos.

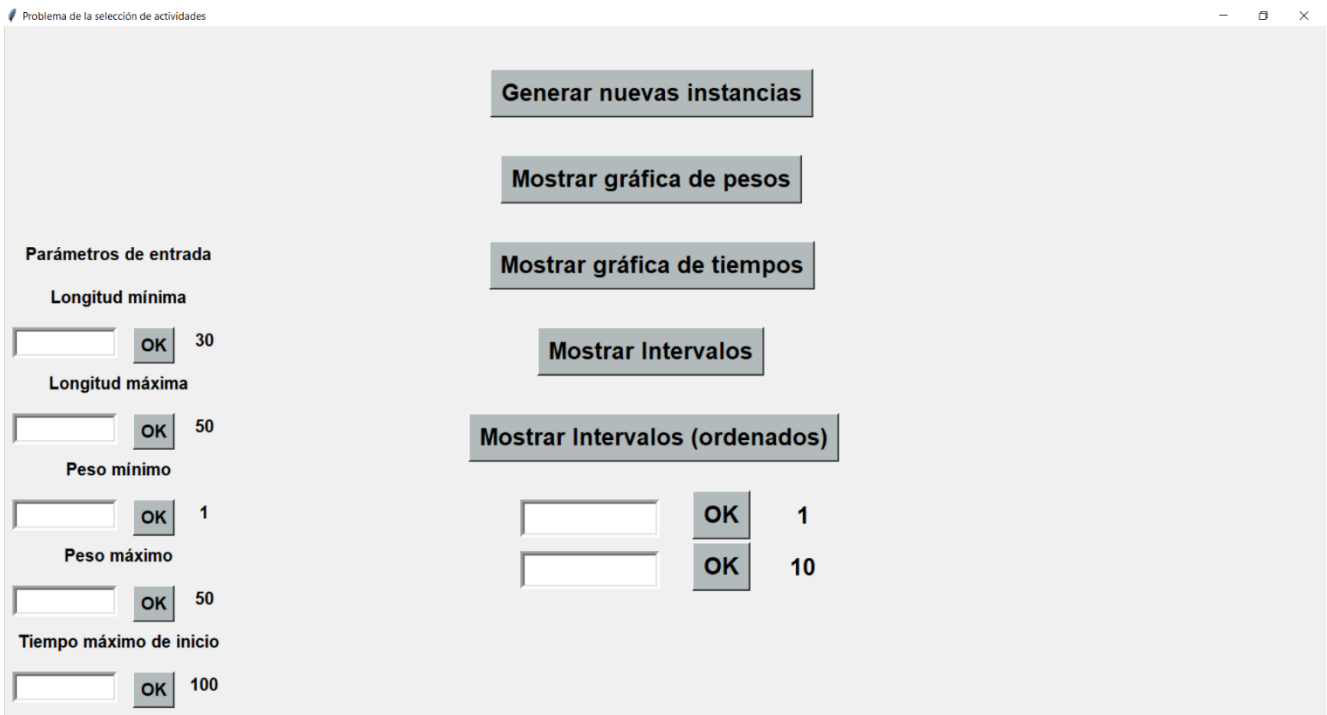


Figura 1. Interfaz de selección de gráficas y cambio de parámetros.

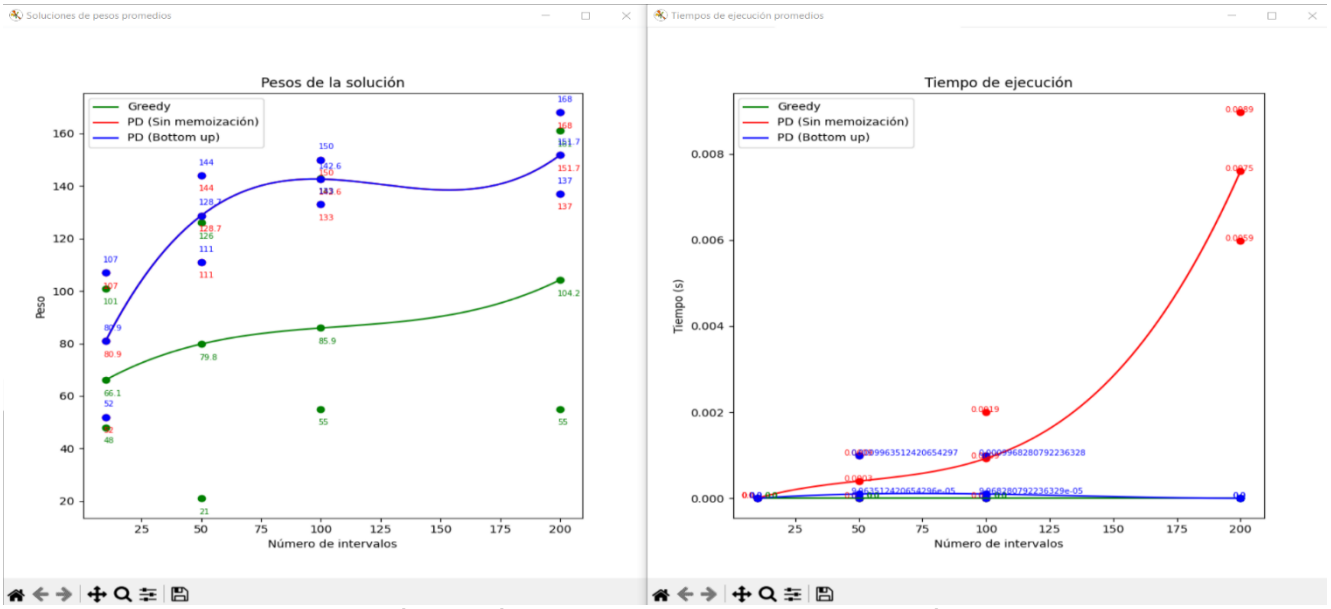


Figura 2. Visualización de gráficas de pesos y tiempos de ejecución de las soluciones.

Como se observa en las gráficas de la Figura 2, se tiene que las curvas representan a los valores promedios de las soluciones tanto en peso como en tiempo de ejecución. Por un lado, los algoritmos de programación dinámica obtuvieron siempre la solución más óptima, mientras que la estrategia *Greedy* obtuvo un resultado aproximado al óptimo, pero no igual. Para la gráfica de tiempos de ejecución cambia el resultado, puesto que el algoritmo *Greedy* y *Bottom up* realizaron el calculó de soluciones en un tiempo cercano a 0, en cambio, las múltiples llamadas recursivas de la estrategia sin memoización generaron que la complejidad de tiempo fuera muy alta y, por tanto, ineficiente. En la Figura 3 se puede apreciar que se obtuvieron resultados muy similares a los de la Figura 2, por lo que la modificación de parámetros de entrada no cambiará el hecho de que Greedy no halla siempre la solución óptima y que no usar memoización ralentiza el rendimiento del algoritmo en función del tiempo.

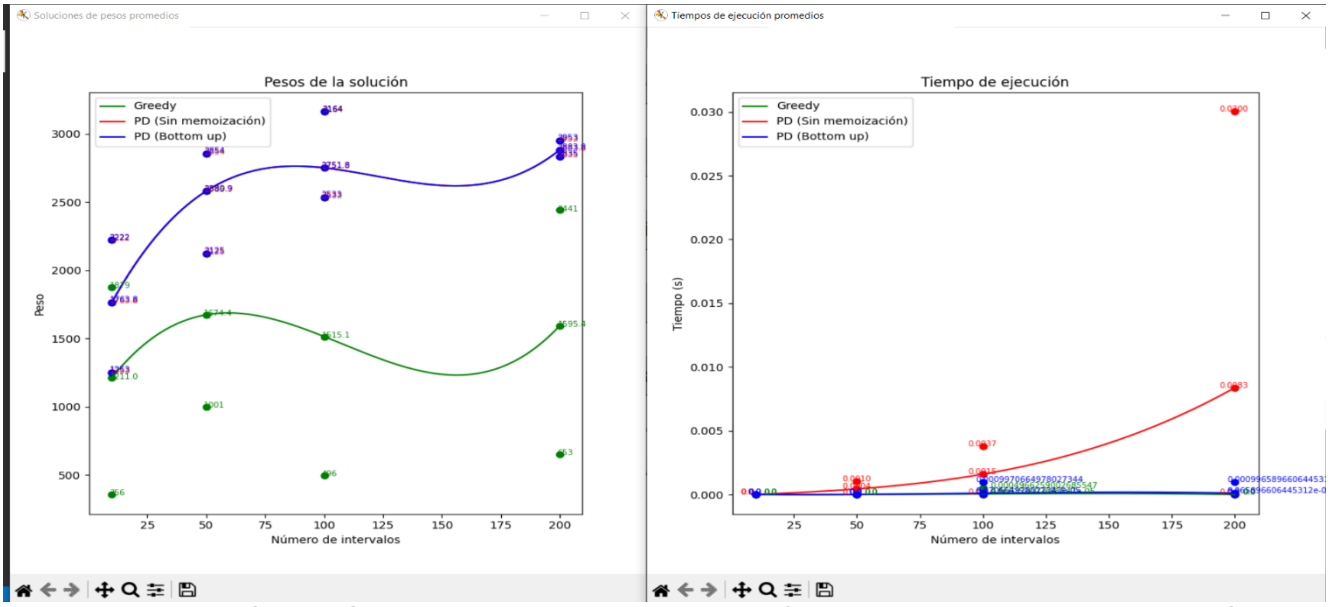


Figura 3. Visualización de gráficas de pesos y tiempos de ejecución de las soluciones con distintos parámetros.

6.1 Complejidad temporal (teórica)

Ahora bien, con respecto a la complejidad temporal teórica se analizarán solamente las funciones utilizadas en los pasos del algoritmo general, debido a que el resultado primordial resulta en la generación de las gráficas, siendo la interfaz de Tkinter, algo que se puede descartar.

Etapa	Complejidad temporal	Explicación
Generador de números pseudoaleatorios (Método de lineal congruencial)	O(1)	En esta etapa se tiene una complejidad constante debido a que la función para generar números pseudoaleatorios no depende de la entrada, puesto que las acciones son constantes ante cualquier tamaño de la entrada.
Generador de instancias	O(n)	En esta etapa se tiene una complejidad lineal gracias a que el número de operaciones dependerá linealmente de la entrada (número de intervalos que se desean). Aunque el algoritmo llama a la función generadora de números pseudoaleatorias, la complejidad sigue siendo la misma, al presentarse complejidad constante.
Ordenamiento por mezcla	O(n log(n))	El ordenamiento por mezcla resulta tener una complejidad temporal de n log(n), debido a que la lista o arreglo que se ordenará, primero se debe dividir en subarreglos del menor tamaño posible (como un árbol binario), lo cual resulta ser un tiempo log(n), y de ahí ordena los subarreglos en un arreglo auxiliar de manera lineal, dando como resultado n*log(n).
Lista de trabajos compatibles	O(n*m)	En esta etapa se tiene una complejidad polinomial, que no llega a ser cuadrática por 2 razones: • El ciclo anidado no empieza desde el elemento inicial, sino desde uno menos, ya que, al buscar intervalos compatibles (no conflictivos), se elimina la evaluación del intervalo consigo mismo. • Al encontrar el primer intervalo compatible, se para el ciclo con entrada m, puesto que se sabe que están ordenados los intervalos, por lo que los demás intervalos a evaluar después de hallar el primero no conflictivo, resultan siempre en no conflictivos.
Algoritmo *Greedy*	O(n)	En esta etapa se tiene complejidad lineal por el hecho de que solo se depende de una entrada, la cual es el número de intervalos de la instancia. El segundo bucle tiene una complejidad lineal también, pero con una entrada más pequeña (conjunto solución *Greedy*). Por lo tanto, se concluye que hay complejidad lineal.
Algoritmo de programación	$O(2^n)$	En esta etapa se tiene un algoritmo de fuerza bruta que utiliza recursividad y que, al no emplear memoización,

dinámica (sin memoización)		provoca que la complejidad aumente de forma exponencial, debido a que cada llamada recursiva aumenta en 2 las siguientes llamadas, repitiendo valores de soluciones.
Algoritmo de programación dinámica (*Bottom up*)	O(n)	En esta etapa se utiliza una lista que almacena los valores de soluciones anteriores, lo que provoca que la complejidad espacial aumente, pero la complejidad temporal resulta lineal porque se evitan repetir procesos gracias al almacenamiento de soluciones anteriores.
Cálculo y almacenamiento de valores promedios, máximos y mínimos	$O(i*k*2^n)$	En esta etapa se tiene una complejidad diferente a las demás, puesto que al ser bucles anidados, pareciera que se trata de una complejidad polinomial, pero las entradas son diferentes, siendo i = 4 y la otra k = 10, por lo que se obtiene O(i*k), no obstante se observa que el peor de los casos y que tiene mayor grado es la solución por el algoritmo de programación dinámica (sin memoización), se concluye que la complejidad es el producto de esta última con O(i*k). Cabe recalcar que, si no se considerará a esta solución ineficiente, la complejidad sería O(i*k*m*n*log(n)).
Graficado de soluciones	O(n)	Aunque no se esta familiarizado con las funciones para graficar y su complejidad temporal, se intuye que, mínimo, el tiempo es lineal por dos motivos: • La función de la curva *Spline* calcula valores de interpolación para generar una curva más suave, por lo que depende de una entrada. • Para graficar el texto (valores promedios, mínimos y máximos) se utiliza un bucle que depende de una entrada, en este caso, de 4 entradas.

6.2 Complejidad temporal (práctica)

Con relación a la complejidad temporal práctica, se empleó el mismo método de medición de tiempo de las soluciones, solo que ahora se aplicó a cada una de las funciones vistas en el apartado anterior.

Etapa	**Tiempo de ejecución**			
	n = 10	**n = 50**	**n = 100**	**n = 200**
Generador de números pseudoaleatorios (Método de lineal congruencial)	0 s	0 s	0 s	0 s
Generador de instancias	0 s	0.0001171 s	0.0029953 s	0.0051121 s
Ordenamiento por mezcla	0.0000654 s	0.0019924 s	0.0029898 s	0.0049869 s
Lista de trabajos compatibles	0 s	0 s	0.0019899s	0.0059913 s
Algoritmo *Greedy*	0 s	0 s	0 s	0 s

Algoritmo de programación dinámica (sin memoización)	0 s	0.0019938 s	0.015796s	0.0874462s
Algoritmo de programación dinámica (*Bottom up*)	0 s	0 s	0 s	0 s
Cálculo y almacenamiento de valores promedios, máximos y mínimos	0 s	0.0029915 s	0.017835 s	0.0986274 s
Graficado de soluciones*	0.1303081 s			

*La gráfica comprende a las 4 entradas.

6.3 Conclusión general

De acuerdo a lo visto en la Figura 1 y 2, junto con los valores de la complejidad teórica y práctica se puede concluir que, para el problema de maximización de pesos del problema de selección de intervalos, la solución *Greedy* resulta ineficiente en términos del peso obtenido, ya que solo halla una aproximación a la solución óptima, no obstante, esa imprecisión la compensa con el tiempo de ejecución, ya que hay algunas situaciones en las que es el más rápido de los 3 algoritmos. Con respecto a la solución de programación dinámica sin memoización, se obtiene el peso óptimo en cualquier escenario, sin embargo, conforme aumenta el número de actividades, tiende a tener una complejidad temporal mayor, provocando una pérdida de tiempo en el sistema. Finalmente, la estrategia *Bottom up* es la mejor de las tres, ya que combina lo mejor de ambos algoritmos, obteniendo el resultado óptimo en todos los casos y teniendo una complejidad de tiempo muy baja. Con esto se indaga que, al diseñar algoritmos se debe tener muy presente la complejidad temporal ante casos de entradas de valores altos, aunque ello requiera sacrificar una parte de la complejidad espacial.

Referencias

[1] T. Yang, H. Feng, C. Yang, Z. Sun y R. Deng, «Towards scheduling to maximize weighted delivered data in maritime CPSs,» *Chinese Control and Decision Conference (CCDC),* pp. 290-295, 2016.

[2] R. Saito y E. Chiba, «A heuristic algorithm for maximizing the total weight of just-in-time jobs under multi-slot conditions,» *IEEE International Conference on Industrial Engineering and Engineering Management,* pp. 84-88, 2016.

[3] Y. Kawamata y S. C. Sung, «On scheduling pseudo just-in-time jobs on single machine,» *The 26th Chinese Control and Decision Conference (2014 CCDC),* pp. 335-340, 2014.

[4] F. Joseph, «Uniform random variate generation with the linear congruential method,» *PANDION: The Osprey Journal of Research and Ideas:,* vol. 1, nº 1, 2020.

[5] J. Erickson, Algorithms, Urbana-Champaign, 2019, pp. 161-164.

[6] K. Moore, J. Khim y E. Ross, «Brilliant.org,» Greedy Algorithms, [En línea]. Available: https://brilliant.org/wiki/greedy-algorithm/. [Último acceso: 1 Diciembre 2022].

[7] D. P. Williamson y D. B. Shmoys, The Design of Approximation Algorithms, New York: Cambridge, 2011.