.UBAfiuba
FACULTAD DE INGENIERÍA

TRABAJO PRÁCTICO 3

TEORÍA DE ALGORITMOS

(75.29 / 95.06 / TB024)

Cuatrimestre: 2º Cuatrimestre 2024

Fecha de entrega: 01 / 12 / 2024

Grupo: 5 (She Don't Give Grafo)

Curso: 03 - Echevarría

Integrantes:

Pablo Choconi	**106388**
Valentín Savarese	**107640**
Gian Luca Spagnolo	**108072**
Brian Céspedes	**108219**
Néstor Palavecino	**108244**

Índice

1. PROBLEMA 1

1.1. Enunciado y Premisas

El problema de **3-Coloreo** es un problema de decisión: dado un grafo $G=(V,E)$, ¿existe una forma de colorear todos los vértices usando a lo sumo 3 colores, tal que cada par de vértices vecinos tengan diferentes colores?. Pero también tiene su contraparte como problema de optimización. Dado un grafo $G=(V,E)$, queremos colorear cada vértice con uno de tres colores, incluso si no se pudiera colorear con colores diferentes cada par de vértices vecinos. Diremos que una arista $e=(u,v)$ será satisfecha si los colores asignados a u y v son diferentes.

Considerar un **3-Coloreo** que maximiza la cantidad de aristas satisfechas, y sea c^* esta cantidad. Desarrollar un **algoritmo aleatorio de tiempo polinomial** que determine un **3-Coloreo** cuyo número esperado de aristas satisfechas sea **al menos 2⁄3 c* aristas.**

1.2. Análisis del Problema

Analizando este enunciado, hemos comenzado analizando problemas antes vistos a lo largo de la cursada. De esta forma, nos podemos encontrar con los problemas de decisión **SAT** y **3-SAT**, los cuales han sido analizados y son considerados como problemas NP-Completos. Este último se basa en la existencia de un conjunto de cláusulas $C_1, C_2, \ldots, C_k$ donde cada una posee tres variables, y existen un conjunto de variables $C=\{x_1, x_2, \ldots, x_n\}$. El problema se basa en intentar conseguir una asignación v de las variables que satisfaga todas las cláusulas simultáneamente.

Teniendo estos problemas en consideración, también se ha planteado a **Max 3-SAT** como un problema de optimización NP-Hard que busca encontrar la asignación v de las variables que maximicen la cantidad de las cláusulas satisfechas. Este algoritmo, en su implementación como algoritmo randomizado, define las X_i como variables aleatorias independientes donde cada Xi tiene un valor booleano y, por ende, la probabilidad de que tomen cualquiera de esos valores es ½, y realizando un análisis profundo del algoritmo, se llega a la conclusión de que el número esperado de cláusulas satisfechas por una asignación aleatoria se encuentra a ⅞ del óptimo, donde la cantidad esperada de corridas del algoritmo para que genere una asignación que satisfaga al menos ⅞ de las cláusulas es $8k$.

Identificado estos problemas correspondientes, se puede establecer una correlación entre el problema de **Max 3-SAT**, que busca cumplir con las restricciones establecidas por las cláusulas con dos valores posibles y donde cada cláusula es independiente de las demás; y el problema de **3-Coloreo**, que busca asignar uno entre 3 colores disponibles a cada vértice, y la elección del color se ve afectada en base a los colores de las demás aristas.

Con respecto al problema de **3-Coloreo**, habiendo realizado el análisis anterior, podemos adaptar el problema antes descrito ya que, en este caso, una arista puede considerarse como satisfecha si y sólo si los vértices extremos extremos tienen colores diferentes, cuya probabilidad de satisfacción es 1 - ⅓ = ⅔ ya que, entre los 3 colores posibles, 1 combinación no satisface.

1.3. Diseño de Nuestra Solución

Una vez realizado el análisis del problema y cómo podemos estructurar nuestra implementación, hemos decidido intentar solucionar este problema basándonos en la siguiente premisa: *"Una arista no puede tener los dos vértices en sus extremos con el mismo color"*, con el objetivo de contabilizar las *Aristas Satisfechas* correspondientes.

Teniendo en cuenta esto, hemos decidido implementar nuestra solución considerando a un **grafo** como un *diccionario*, cuya *clave* es el número del vértice y su *valor* está compuesto por una lista con los vértices adyacentes, de modo que consideramos a una arista como aquellos pares de vértices donde ambos están presentes en sus listas correspondientes. Al tratarse de un problema que utiliza un grafo **no dirigido** y **sin pesos**, el problema se ve bastante simplificado, aunque hay que tener en consideración que cada arista estará presente dos veces en el grafo (una vez por cada vértice correspondiente), por lo que hay que dividir siempre la cantidad de aristas al manejar este tipo de grafos.

Habiendo implementado este algoritmo con este tipo de datos simple para asemejar a un grafo, la implementación resulta evidente y bastante sencilla, ya que simplemente hay que recorrer cada uno de los vértices de un grafo, analizando los vértices adyacentes y comparando los colores correspondientes para determinar si una arista es satisfecha o no.

Sin embargo, para este problema, hay que tener en cuenta la cantidad de aristas que posee cada vértice debido a que, si esta cantidad es demasiado alta en comparación con la cantidad de vértices del grafo, puede que nunca se pueda encontrar una solución óptima con una cantidad de aristas satisfechas mayor o igual al ⅔ de las aristas totales.

1.4. Pseudocódigo

```
problema_3_coloreo_aleatorio(grafo, limite):

    # El grafo es un diccionario cuyas claves representan los índices de cada vértice, y los valores están compuestos por
    # listas con aquellos vértices adyacentes.
    # El límite es un número entero que corresponde a la cantidad mínima de aristas satisfechas necesarias.
    # La cantidad mínima es ⅔ de la cantidad total de aristas.

    grafo_coloreado = diccionario
    aristas_satisfechas = 0

    MIENTRAS QUE la cantidad de aristas satisfechas sea menor al límite previamente calculado:

        # Se intenta colorear un nuevo grafo, para validar si se trata de una respuesta correcta.
        nuevo_grafo_coloreado = diccionario
        PARA cada vertice de grafo:
            rellenar con un color aleatorio en nuevo_grafo_coloreado[vertice] de entre las 3 opciones posibles

        # Ahora, se calcula la cantidad de aristas satisfechas en esta nueva iteración del ciclo.
        nuevo_aristas_satisfechas = 0
        PARA cada vertice de grafo:
            PARA cada vertice_adyacente de grafo[vertice]:
                SI el color de nuevo_grafo_coloreado[vertice] != nuevo_grafo_coloreado[vertice_adyacente]:
                    nuevo_aristas_satisfechas += 1
```

Se divide a la mitad la cantidad de aristas satisfechas ya que se trata de un grafo no dirigido
nuevo_aristas_satisfechas /= 2

Se verifica si este nuevo grafo coloreado es mejor que el grafo coloreado anteriormente
SI la cantidad de **nuevo_aristas_satisfechas** > **aristas_satsfechas**:
grafo_coloreado = **nuevo_grafo_coloreado**
aristas_satisfechas = **nuevo_aristas_satisfechas**

RETORNAR **grafo_coloreado**, **aristas_satisfechas**

1.5. Implementación

```python
POSIBLES_COLORES = [1, 2, 3]

def problema_3_coloreo_aleatorio(grafo, limite):
    mejor_asignacion = {}
    max_aristas_satisfechas = 0
    cantidad_iteraciones = 0

    while max_aristas_satisfechas < limite:
        asignacion = {}
        aristas_satisfechas = 0
        cantidad_iteraciones += 1

        for v in grafo:
            asignacion[v] = random.choice(POSIBLES_COLORES)

        for v in grafo:
            for u in grafo[v]:
                if asignacion[v] != asignacion[u]:
                    aristas_satisfechas += 1
        aristas_satisfechas //= 2

        if aristas_satisfechas > max_aristas_satisfechas:
            mejor_asignacion = asignacion
            max_aristas_satisfechas = aristas_satisfechas

    return mejor_asignacion, max_aristas_satisfechas, cantidad_iteraciones
```

1.6. Complejidades

Analizando este algoritmo, la **Complejidad Temporal** es calculada de la siguiente manera:

- **Complejidad Temporal** de asignación de colores en el grafo: $O(n)$
- **Complejidad Temporal** de verificación de aristas satisfechas: $O(n) * O(v)$
- **Complejidad Temporal** de iteración hasta encontrar solución óptima: $O(i)$

Complejidad Final Temporal: $O(i) \cdot (O(n) + O(n) \cdot O(v)) = O(i) \cdot O(n) \cdot O(v)$

Complejidad Espacial (correspondiente al diccionario del grafo): $O(n) \cdot O(v)$

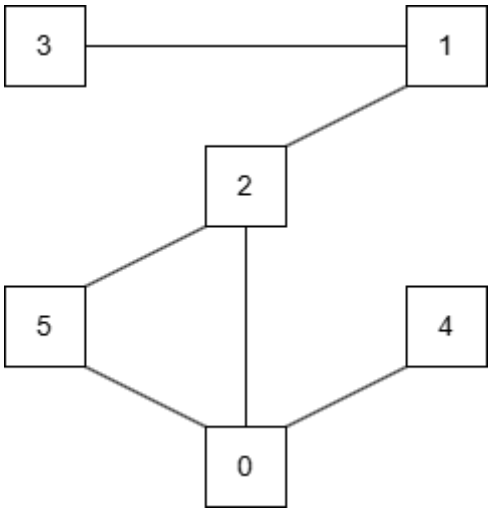
1.7. Seguimiento y Resultados de Ejecución

Una vez habiendo explicado la implementación del algoritmo, podemos ponerlo a prueba con unos conjuntos de datos específicos y reducidos. Nuestra implementación corresponde al archivo `ejercicio_1.py` y ademas disponemos de archivos de un archivo de testeo `ejercicio_1_test.py` el cual puede recibir un grafo cargado en un archivo .csv creado a gusto por el usuario, y puede poner a prueba la asignación de colores aleatoria que hemos implementado. Los comandos correspondientes, junto al funcionamiento en detalle de cada archivo, están presentes en el `README` adjunto a la entrega.

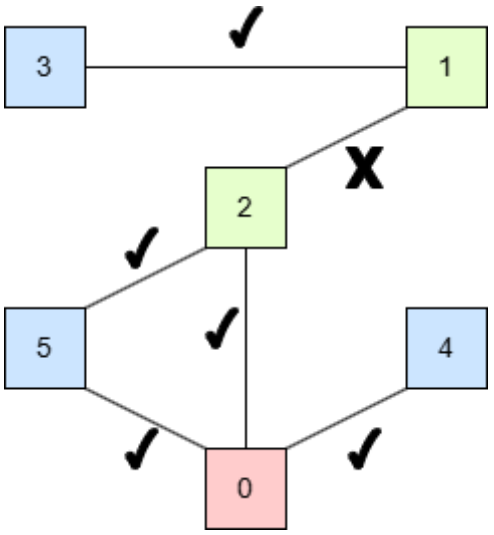
Sin embargo, disponemos de antemano de 3 grafos creados previamente para realizar tests específicos. En la siguiente imagen podemos observar la ejecución del grafo presente en el archivo `grafo_1.py`:

```
adduser@GianLucaSpagnolo:~/tda/repo/tp3$ python ej1/ejercicio_1_test.py
Problema de 3-Coloreo

Test con archivo: ej1/res/grafo_1.csv

Tiempo de ejecución: 0.00001120 segundos
Grafo obtenido (con colores de cada vertice): {0: 1, 1: 2, 2: 2, 3: 3, 4: 3, 5: 3}

Aristas Totales: 6
Aristas Satisfechas: 5
Cantidad de Iteraciones: 1
```

Esta ejecución corresponde al gráfico visualizado en la imagen a continuación, con un total de 6 aristas y 6 vértices.



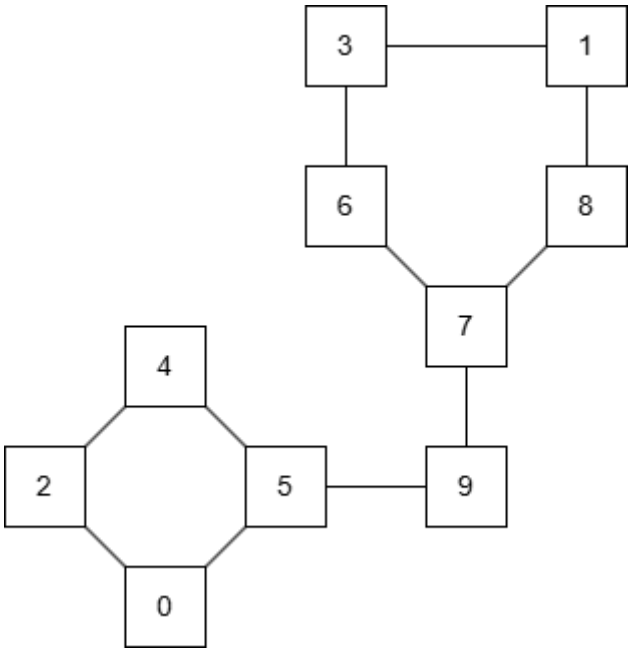
Viendo los resultados de la ejecución aleatoria anterior, solo basto una iteración para conseguir, de forma aleatoria, una cantidad de aristas satisfechas mayor a ⅔ de la cantidad total de aristas, como se ha establecido previamente. Teniendo 5 aristas satisfechas, y habiendo necesitado una única iteración, el **tiempo de ejecución** ha sido bastante reducido a diferencia de haber probado con un grafo de mayor tamaño. A continuación, se puede visualizar el grafo obtenido como resultado:



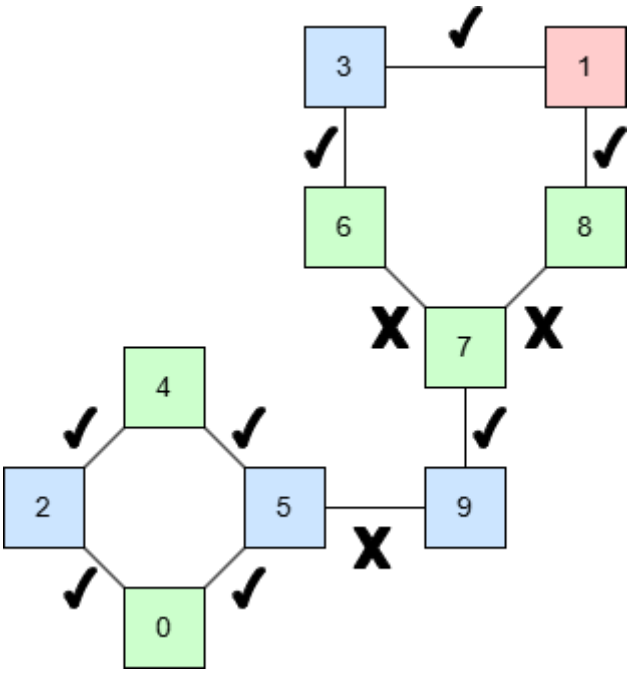
A continuación, mostraremos la ejecución de nuestro algoritmo utilizando el grafo presente en el archivo `grafo_3.py`, el cual tiene un total de 10 vértices y 11 aristas, para poder observar la ejecución del algoritmo con un grafo un poco mayor:

```
adduser@GianLucaSpagnolo:~/tda/repo/tp3$ python ej1/ejercicio_1_test.py
Problema de 3-Coloreo

Test con archivo: ej1/res/grafo_2.csv

Tiempo de ejecución: 0.00002070 segundos
Grafo obtenido (con colores de cada vertice): {0: 3, 1: 1, 2: 2, 3: 2, 4: 3, 5: 2, 6: 3, 7: 3, 8: 3, 9: 2}

Aristas Totales: 11
Aristas Satisfechas: 8
Cantidad de Iteraciones: 3
```

Esta ejecución ha necesitado de una cantidad mayor de iteraciones para obtener un grafo coloreado correcto, y que cumpla con la cantidad mínima de aristas satisfechas establecidas. Es por eso que el **tiempo de ejecución** ha sido mayor al anterior grafo analizado. El gráfico utilizado en esta ejecución se puede observar en la siguiente imagen:



Y, posteriormente, podemos analizar a detalle los colores aplicados en cada vértice, y como las aristas satisfechas corresponden a una cantidad óptima tal cual como lo pide el enunciado de este ejercicio:



De esta forma, es posible probar la ejecución de nuestra implementación con diferentes conjuntos de datos correspondientes a grafos. Este análisis permite observar un leve incremento en el **tiempo de ejecución** necesario para ejecutar grafos mayores, que puedan necesitar de una mayor cantidad de iteraciones, hasta encontrar el grafo coloreado óptimo.

Sin embargo, hay que tener en cuenta que, a medida que incrementa la cantidad de aristas del grafo, se vuelve más difícil encontrar una solución óptima en una menor cantidad de operaciones debido a que la cantidad de colores a aplicar en cada vértice es siempre permanente: 3. Es por esto que, si bien este algoritmo puede manejar grafos de tamaños variables y mucho más grandes, si la cantidad de aristas excede abruptamente la esperada, el **tiempo de ejecución** incrementará acordemente hasta encontrar la solución óptima en caso de que sea posible.

2. PROBLEMA 2

2.1. Enunciado y Premisas

En el **Problema de la Mochila** tenemos `n` ítems, cada uno con un peso `wi` y un valor `vi`. También tenemos un límite máximo `W` para el peso total de la mochila. El problema consiste en encontrar un subconjunto `S` de ítems que maximice el valor total `V` sujeto a la capacidad `W` de la mochila. En clase vimos que podemos diseñar un algoritmo que seleccione de manera aproximada los elementos del conjunto `S` cumpliendo con la capacidad `W` de la mochila, y que se aproxime al valor máximo óptimo con un margen `V/(1+ε)`. Pero en la vida real uno siempre quiere poner algo más dentro de la mochila, haciendo un poco de fuerza para cerrarla. Para un valor fijo `ε>0`, implementar un **algoritmo de aproximación** que encuentre un subconjunto de ítems `S` que logre un valor de al menos `V` pero que se permita exceder la capacidad de la mochila en `(1+ε)W`. El algoritmo diseñado debe ser de **tiempo polinomial**.

2.2. Diseño y Estructuras de Datos

Las únicas estructuras de datos utilizadas son arreglos y matrices (implementadas como arreglos de arreglos).

El algoritmo es una implementación del algoritmo para resolver el **Problema de la Mochila por Aproximación** y **Programación Dinámica** detallado en el libro de *Algorithm Design - Jon Kleinberg and Eva Tardos*. Los únicos cambios relevantes respecto al algoritmo del libro son:

- Reconstrucción de la solución. Es decir el armado del arreglo con los elementos seleccionados.
- En lugar de usar la capacidad *W* que se recibe para encontrar el máximo valor posible después de armar la matriz se utiliza *(1+ε) W* para cumplir con la condición del enunciado.

2.3. Pseudocódigo

```
llenar_mochila(pesos, valores, capacidad, epsilon):

    b = epsilon / (2 * n) * v_star

    valores_escalados = techo(valor / b) para cada valor en valores

    valor_maximo_posible = sum(valores_escalados)

    matriz = matriz de (n + 1) x (valor_maximo_posible + 1) con todos los valores en 0

    PARA i = 1 hasta n
      PARA v = 0 hasta valor_maximo_posible
        SI v > sum(valores_escalados[1..i])
          matriz[i][v] = matriz[i-1][v]
```

```
        SINO
            matriz[i][v] = max(matriz[i-1][v], matriz[i-1][v - valores_escalados[i]] + valores[i])

capacidadExtendida = capacidad * (1 + epsilon)

indice_de_solucion = 0
PARA j = valor_maximo_posible hasta 0 con incremento -1
    SI matriz[n][j] <= capacidadExtendida
        indice_de_solucion = j
        BREAK

valores_seleccionados = []

PARA i = n hasta 1 con decremento -1
    SI indice_de_solucion <= 0
        BREAK
    SI matriz[i][indice_de_solucion] != matriz[i-1][indice_de_solucion]
        valores_seleccionados.agregar(i)
        indice_de_solucion = indice_de_solucion - valores_escalados[i]

RETORNAR valores_seleccionados
```

2.4. Implementación

```
def knapsack_approx(weights, values, capacity, epsilon):
    n = len(values)

    # Calcular el valor máximo v* (máximo de los valores)
    v_star = max(values)

    # Calcular el parámetro de redondeo b
    b = epsilon / (2 * n) * v_star

    # Redondear los valores de los elementos
    scaled_values = [int(math.ceil(value / b)) for value in values]  # Escalamos los
valores

    # Resolver el problema de la mochila usando el algoritmo de programación dinámica con
los valores escalados
    max_value = sum(scaled_values)
    matrix = [[0] * (max_value + 1) for _ in range(n + 1)]



    scaled_values_current_sum = 0
    # Llenar la matriz M
    for i in range(1, n + 1):
        scaled_values_current_sum += scaled_values[i - 1]
        for j in range(1, max_value + 1):
            if j > scaled_values_current_sum:  # Si el valor excede la suma de los
primeros i elementos
                matrix[i][j] = weights[i - 1] + matrix[i - 1][j]
            else:
                matrix[i][j] = min(matrix[i - 1][j], weights[i - 1] + matrix[i - 1][max(0,
j - scaled_values[i - 1])])

    # Reconstrucción de la solución

    extraCapacity = capacity * (1 + epsilon) # Según la consigna se puede exceder la
capacidad en funcion de epsilon

    solutionIndex = 0
    for j in range(max_value, 0, -1):
        if matrix[n][j] <= extraCapacity:
            solutionIndex = j
```

```
            break

    selectedItems = []
    totalWeight = 0
    totalValue = 0

    for i in range(n, 0, -1):
        if solutionIndex <= 0:
            break
        # Si el peso del elemento actual es diferente al peso del elemento anterior
significa que el elemento actual fue seleccionado
        if matrix[i][solutionIndex] != matrix[i - 1][solutionIndex]:
            selectedItems.append(i)
            totalWeight += weights[i - 1]
            totalValue += values[i - 1]
            solutionIndex -= scaled_values[i - 1]

    result = {
        "selectedItems": selectedItems,
        "totalValue": totalValue,
        "totalWeight": totalWeight
    }

    return result
```

2.5. Seguimiento

Datos de entrada:

- Pesos de los ítems: [1, 7, 4, 5, 6]
- Valores de los ítems: [30, 21, 35, 48, 40]
- Capacidad máxima de la mochila: 15
- Parámetro de precisión epsilon (ε): 0.5

Pasos que sigue el algoritmo:

1. Se calcula b = 2.4b
2. Se arma un arreglo de los valores escalados: [13, 9, 15, 20, 17].
3. Calcula la sumatoria de los valores escalados (máximo valor escalado) = 74.
4. Construye la matriz.
5. Como itera en base al número de elementos y a la sumatoria de los valores, el ciclo anidado itera 370 veces (74 × 5) y no hay una contribución significativa a la solución

final dentro de cada una de estas iteraciones individuales. Se procede a explicar de forma general qué haría cada iteración por practicidad:

- Itera con i=1,…,n y por cada iteración:
  - Suma el valor del item actual a la suma de todos los valores de los items evaluados (scaled_values_current_sum)
  - itera con j=1,...., máximo valor escalado y en cada iteración evalúa cuál es la opción con menor peso que cumple con el valor que se busca alcanzar:
    - Si j > scaled_values_current_sum, significa que el valor que se busca es imposible de alcanzar. Agrega el ítem a la solución y continúa.
    - En caso contrario, evalúa qué opción alcanza el objetivo con el menor peso posible:
      - Opción 1: Se mantiene la solución anterior.
      - Opción 2: Se le suma el ítem actual a la solución anterior.

6. Se calcula la capacidad extendida: (1 + 0.5) * 15 = 22,5
7. Se busca el índice de la solución iterando la última fila de la matriz de manera inversa. Como los valores más grandes están al final, la primera solución que se encuentre que cumpla con la condición definida por la capacidad extendida es la solución válida.
8. Se reconstruye la solución final iterando de forma inversa la matriz utilizando el índice de la solución. Evaluando los pesos de la solución parcial actual con la anterior. Cuando esta es diferente, significa que el ítem fue seleccionado.
9. Se retorna la solución final.

Resultado:

- **Ítems seleccionados:**
  - Ítem 5 - Peso: 6 - Valor: 40
  - Ítem 4 - Peso: 5 - Valor: 48
  - Ítem 3 - Peso: 4 - Valor: 35
  - Ítem 1 - Peso: 1 - Valor: 30
- **Totales:**
  Valor total: 153
  Peso total: 16

2.6. Complejidades

El algoritmo puede reducirse a los siguientes pasos:

- Búsqueda de valor máximo *v** de *A* siendo *A* los ítems que se recibe como input. **Complejidad:** *O(n)*
- Cálculo de b **Complejidad:** *O(1)*

- Crear arreglo de valores escalados *Â*. **Complejidad:** *O(n)*
- Calcular sumatoria de valores escalados C **Complejidad:** *O(n)*
- Construir matriz con tamaño *n * C*. **Complejidad:** *O(n * C)*
- Llenar la matriz. Se realiza un ciclo con *n* iteraciones y en cada iteración se realiza un ciclo con sumatoria de iteraciones. **Complejidad:** *O(n*C)*
- Cálculo de la nueva capacidad. **Complejidad:** *O(1)*
- Búsqueda del índice donde termina la solución. **Complejidad:** *O(C)*
- Reconstrucción de la solución **Complejidad:** *O(n*C)*

Lo que nos deja con la ecuación:

$$O(n)+O(1)+O(n)+O(n)+O(n\times C)+O(n\times C)+O(n\times C)+O(1)+O(C)+O(n\times C) = O(n\times C)$$

Si tenemos un epsilon para el cual al calcular b este es entero y los valores escalados son los valores de A redondeados a un múltiplo de b y divididos por b entonces sabemos que con *i=1,....,n*

$$\hat{v}_i \leq v_i$$

Siendo *v̂i* el valor escalado de *vi* .

A partir de esto podemos inferir que el valor más grande de *A* es más grande que el valor más grande de *Â* para i=1,....,n

$$\hat{v}_i \leq v^*$$

Y

$$\sum \hat{v}_i \leq n \cdot v^*$$

Por lo que podemos tomar $n \cdot v^*$ como cota superior y reemplazar C en la ecuación de complejidad por lo que se puede decir que **la complejidad es *O(n * n * v*) = O(n^2 * v*)*** por lo que el algoritmo aproximado es pseudo-polinomial y depende del valor más grande que se recibe como input.

Si el epsilon es suficientemente pequeño como para que al calcular b suceda que $0 < b < 1$ entonces los valores escalados serán más grandes que los valores originales y como la sumatoria de estos se usa para iterar aumentará el tiempo de ejecución.

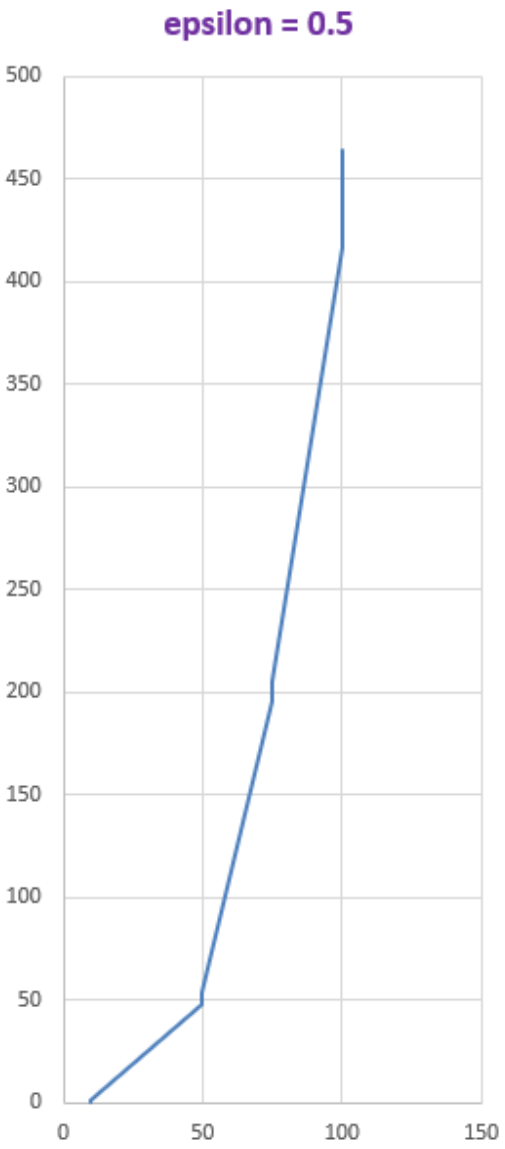
2.7. Tiempos de Ejecución

Como se puede ver en los gráficos. El tiempo de ejecución aumenta mientras más elementos hay y mientras más cerca a 0 esta epsilon.

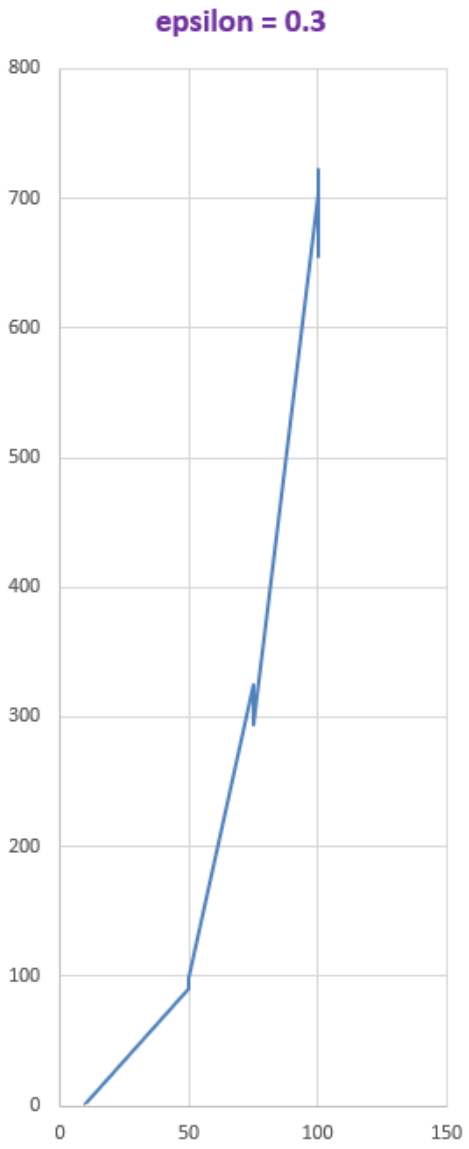
Esto tiene sentido considerando que la parte que más tiempo consume del algoritmo es donde se llena la matriz y está definida por dos ciclos for. Uno que itera por cantidad de elementos y el otro por la sumatoria de los valores reescalados. Por lo que uno podría suponer que:

- A mayor cantidad de elementos mayor cantidad de iteraciones.
- Como los valores reescalados se calculan dividiendo los valores originales por b y en el cálculo de b, epsilon es el numerador. Mientras más pequeño sea el epsilon, más pequeño será b y mientras más pequeño sea b más grande será el valor reescalado y por consiguiente su sumatoria.

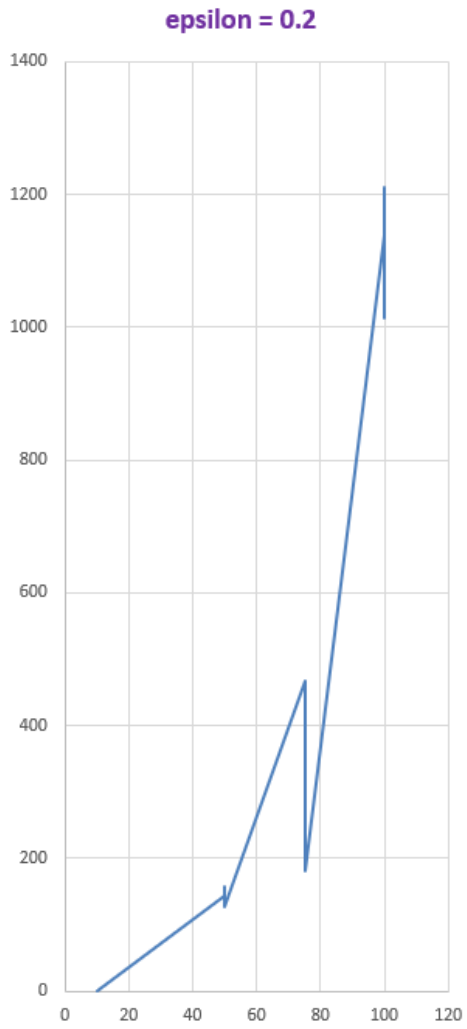
Ambos puntos están confirmados por la información que se puede ver en los gráficos.



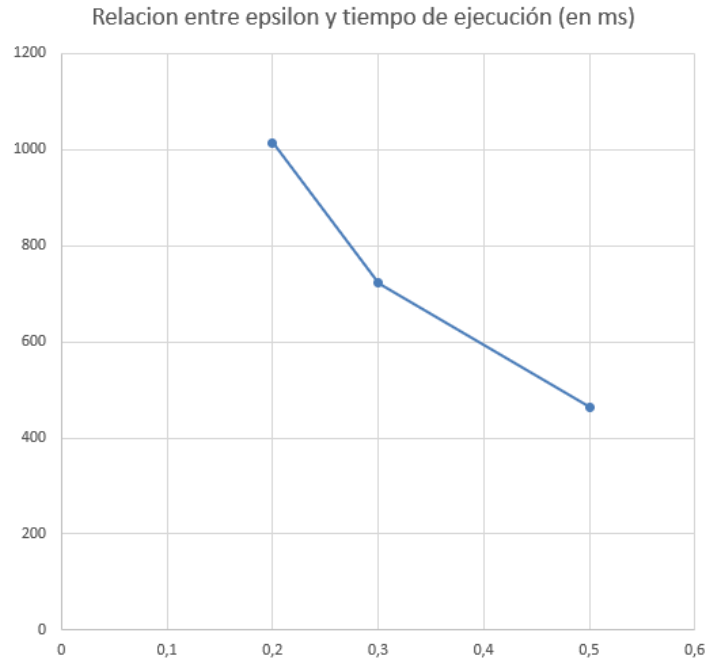
Relación entre numero de elementos y tiempo de ejecución en ms con epsilon = 0.5



Relación entre numero de elementos y tiempo de ejecución en ms con epsilon = 0.3



Relación entre numero de elementos y tiempo de ejecución en ms con epsilon = 0.2



Relación entre epsilon y el tiempo de ejecución en ms.

REFERENCIAS

Algorithm Design. (2005). Jon Kleinberg and Eva Tardos. Tsinghua University Press.

Knapsack Problems: Algorithms and Computer Implementations. (1990). Silvano Martello and Paolo Toth. DEIS, University of Bologna.