DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Eligiendo justito

7 de octubre de 2018 — Algoritmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Buceta, Diego	001/17	diegobuceta35@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

1.1. El problema y sus análisis

El subset sum problem o problema de la suma de conjunto es un problema muy conocido dentro de las ciencias de la computación y matemática por ser un objeto de estudio dentro de la clase NP-problem [1], se destaca por su facilidad para describirlo y gran dificultad para resolverlo. Los algoritmos más conocidos que pueden resolverlo tienen una complejidad temporal exponencial y esto se traduce en imposibilidad de resolverlo en casos donde la cantidad de elementos sea grande.

Además de las diferentes versiones de este problema [2] existen usos muy diversificados en el ámbito cotidiano y científico. No solo sirve como fuente de nuevas investigaciones en el campo de la teoría de complejidad sino también en ramas como la criptografía, programación genética, etc[3][4]. Imaginemos un algoritmo que resuelva problemas a modo de entrenamiento, problemas de tamaños variados y que en muchos casos es necesario clasificar las diferentes soluciones. En los algoritmos genéticos se suele desarrollar un algoritmo que funcione o se comporte como un gen y evaluarlo en consecuencia. Existen casos en los que el output no tienen una correlación real con un gen y es necesario técnicas de apartado o filtrado obtener los más aproximados, y una de ellas está fuertemente relacionada con el Subset Sum Problem.

Definición: Dado C conjunto de enteros no negativos y V entero no negativo se desea saber si existe un s $\subseteq$ S tal que $\sum_{r \in R} r = V$, y si existe que tenga la mínima cardinalidad.

Intuición Una de las primeras aproximaciones para buscar una solución puede ser la de formar todos los conjuntos posibles con los elementos de S y analizar si alguno suma V. Por un lado, este algoritmo asegura que si existe solución se va a encontrar; por el otro lado y como contrapartida de lo anterior, si no existe solución o S es muy grande la cantidad de pruebas que son necesarias superará las capacidades de cómputo actuales. En la actualidad hay diversas ramas que se encuentran con estas dificultades, y la manera en que se suelen evitar es utilizando aproximaciones para el problema y no algoritmos exactos[5]. reducir esto tampoco tiene sentido la suposición de determinadas características de S, como el orden de sus elementos, dado que siempre se verificarán todos los subconjuntos.

Otras formas Los análisis previos arrojan suficientes razones como para ahondar en otras soluciones.

Una algoritmo publicado por Horowitz Ellis y Sahni Sartaj [6][7] consiste en dividir los n elementos en dos conjuntos separados, cada uno con $\frac{n}{2}$ elementos. Para cada uno de los conjuntos se almacenan en dos listas la suma de los posibles subconjuntos junto con su cardinal. Posteriormente se utiliza algún algoritmo de comparaciones para ordenarlas, L_1 en orden decreciente y L_2 creciente, lo que suma una complejidad temporal de $2 \cdot O(2^{\frac{n}{2}} \cdot \frac{n}{2})$. Con las dos listas ordenadas, la idea principal es tomar un elemento de cada lista y compararlo con V. Por un lado se toma el elemento más grande (primero)de L_1 y el más chico (primero)de L_2. Entonces:

- Si la suma supera a V: El algoritmo avanza un elemento en L_1
- Si la suma es menor a V: El algoritmo avanza un elementom en L_2
- Si la suma coincide con V: se muestra la solución.

2. Algoritos presentados

En este trabajo presentaremos diferentes algoritmos con los que buscaremos experimentar y encontrar los contextos en los cuales sería beneficioso la aplicación de alguno de ellos.

2.1. Fuerza Bruta

Esta forma de resolver el problema está fuertemente relacionada con las primeras técnicas que surgieron para el problema de la suma de subconjuntos. Lo primero que debe hacerse es definir un dominio de posibles soluciones del problema, y posteriormente describir un algoritmo que permita alcanzarlas a todas.

En nuestro problema, vamos a definir nuestro espectro de posibles soluciones al conjunto de partes del conjunto inicial, S. Este conjunto estará formado por todos los posibles subconjuntos que se puedan tomar con elementos de S. Dado que cada elemento tiene dos opciones, estar o no en un determinado subconjunto, las opciones entonces son: $2 \cdot 2 \cdot 2 \cdots 2 = 2^n$

[1] Problemas de decisión y optimización donde se busca encontrar si existe solución o si existe una mejor que la conocida

[2] Problema de la carga de la mochila; Problema de selección de conjuntos de un conjunto, etc

2.2. Algoritmo

```
FuerzaBruta(Conjunto(enteros) : C, entero : V)
1  i ← 0
2  P ← generarPartes(C)
3  sol ← −1
4  n ← Tam(C)
5  repeat
6          suma ← sumaElementos(P, i)
7          card ← cantElementos(P, i)
8          if suma == V then
9                  if sol == −1 then
10                         sol ← card
11                 else
12                         if sol > card then
13                                 sol ← card
14 until i <= 2^n
```

Función: generarPartes El código fuente del problema genera el conjunto de partes de la siguiente manera: una variable i se mueve entre 0 y 2^n - 1, y en cada uno de esos valores analiza su representación en bits y además otra variable j se mueve entre 0 y n-1. Es decir, cada vez que i actualiza su valor, j se actualiza n veces. En esas n comparaciones lo que se hace es analizar si el bit jésimo se corresponde un bit marcado en uno de i. Si esto pasa, significa que en el subconjunto iésimo debe estar el elemento j, y así para cada elemento. Esto se hace mediante el corrimiento de bits en O(1) que ofrece $C++$.

Función: sumaElementos y cantElementos Dado un conjunto de conjuntos y un valor, una retorna la suma y la otra el cardinal del subconjunto i-ésimo. A modo de una mejor visualización de las ideas del algoritmo estas funciones cumplen un objetivo simbólico y no tienen una correspondencia simétrica con el código, en caso de plasmar este pseudocódigo en un lenguaje, simplemente cambiará la complejidad espacial debido a que habría 2^n subconjuntos almacenados.

2.2.1. Complejidad

Desde el punto de vista de complejidad, si se define en función del tamaño de entrada, y tomando n como el tamaño de S, nuestro algoritmo será $\theta(n \cdot 2^n)$, porque se recorrerá cada subconjunto del conjunto de partes de S y para cada uno revisará cuáles de los n elementos de S están en ese subconjunto.

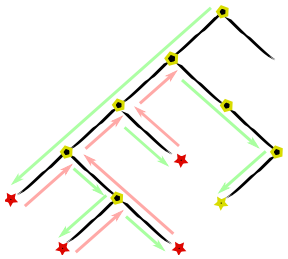
2.3. Backtracking

Lo que caracteriza a esta forma de resolver el problema es que vamos a buscar definir situaciones en las que seguir computando la solución no tenga sentido. Esto quiere decir abandonando posibles candidatos a solución por temás de optimalidad o posibilidad de la solución, a esto se lo denomina podas.

2.3.1. Podas

- Por optimalidad: Si en el proceso de computo de una posible solución, se llegara al caso en que el subconjunto de elementos actual contiene más elementos que la mejor solución encontrada hasta ahora, entonces no se buscará y computará ninguno de los subconjuntos que surjan de tener como los primeros k elementos del subconjunto actual y siendo k el tamaño de éste.
- Por factibilidad: Si en el proceso de cómputo de una posible solución, se llegara al caso en que la sumatoria de los elementos del conjunto formado actual supera a V, significa que no tiene sentido probar con a ninguno de los elementos hacia la derecha, porque están ordenados de forma creciente, y ninguno de los conjuntos que surjan de combinar los elementos restantes de S con el conjunto actual van a funcionar.

La siguiente imagen muestra la idea de las podas. Si tomamos el punto amarillo superior como el inicio de una parte de los posibles caminos del problema total vemos cómo las flechas verdes marcan los caminos que se siguieron por cada nodo y cómo al llegar a puntos problemáticos las flechas rojas marcan el retroceso y recorrido de otro camino, hasta agotarlos a todos. Los puntos problemáticos simbolizan otros inicios de conjuntos de caminos que por algún motivo se decide 'podarlos' y no analizarlos en profundidad.



[3]

2.3.2. Algoritmo

```
BACKTRACKING(Conjunto(enteros) : C, entero : V)
 1  sol ← −1
 2  if V = 0 then
 3          sol ← 0
 4          Devolver 0
 5  OrdenarAsc(c)
 6  sol ← auxBacktracking(c, 0, V, sol, 0, 0)
 7  if sol = −1 or sol >|C| then
 8          Devolver − 1
 9  else
10          Devolver sol
```

```
AUXBACKTRACKING(Conjunto(enteros) : C, entero : sol, entero : V, entero : desde, entero : sumaElem, entero : cantElem)
 1  if sumaActual >valorObjetivo then
 2          Devolver |C| + 1
 3  if sol! = −1 and cantElem >= sol then
 4          Devolver |C| + 1
 5  if sumaActual = valorObjetivo then
 6          sol ← cantElem
 7          Devolver 0
 8  if desde < |C| then
 9          if c[desde] + sumaActual >valorObjetivo then
10                  Devolver |C| + 1
11          else
12                  valor1 ← auxBackTracking(c, desde + 1, valorObjetivo, sol, sumaActual, cantElem)
13                  valor2 ← 1 + auxBacktracking(c, desde + 1, valorObjetivo, sol, sumaActual + c[desde], cantElem + 1)
14                  Devolver min(valor1, valor2)
```

Detalle

La función principal es Backtracking, que se encarga de recibir el conjunto con el que trabajar, el valor objetivo, y prepara lo necesario para resolver el problema con metodología de Backtracking. En nuestro caso, ordena el conjunto, según documentación de C++, en O($n \cdot \log n$) e inicializa los flags que usará la función recursiva. Estos son: sumaActual, que contiene la suma de los elementos tomados como posible solución por una determinada rama; cantElem que contiene la cantidad de elementos tomados como posible solución por una determinada rama; desde, que permite que cada llamado recursivo sepa desde donde a agregar elementos a la solución en una determinada rama; sol, que es donde se almacena la mejor solución actual que se encontró para el problema y que cada llamado recursivo revisa para conocer si es posible que esa rama mejore la solución actual y en caso contrario podarla. Esta variable, al igual que los conjuntos, se utilizan mediante referencias para no hacer copias de C que puede ser excesivamente grande y para poder conocer en cada subproblema la solución encontrada actualmente, que pudo haber sido actualizada por una rama posterior a la que generó este llamado recursivo y de esta forma puede tener acceso a la solución actual.

2.3.3. Complejidad

Respecto a la complejidad temporal y estableciéndola en función de tamaño de la entrada, |S| = n, veamos que el peor caso sería si para ningún llamado recursivo existe poda, es decir, el valor objetivo es mayor que cualquier sumatoria de subconjunto de C (no hay poda de factibilidad) y como tampoco se encuentra solución alguna no se puede podar llamados recursivos por poda de optimalidad. De modo que terminaría después de probar con cada subconjunto (el conjunto de partes de C).

[3] Imagen tomada del blog de Steve Pemberton

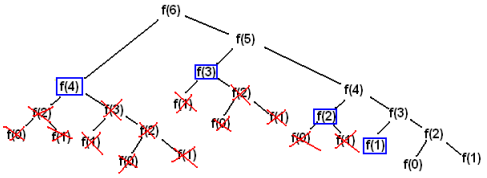
Cada elemento de C (n en total), generaría dos llamados recursivos con el flag desde = desde + 1, o sea que estos llamados recursivos del elemento C_1 generarían a su vez cada uno de ellos otros dos llamados, pero de tamaño |C|-1 el conjunto y así sucesivamente. Esto generaría un árbol binario completo de 2^n nodos y n hojas. Como cada llamado recursivo va actualizando la suma de los elementos que le llega de su predecesor, no aporta a la complejidad (más alla de los 2^n nodos que se recorren) la suma de los elementos que se consideran en esa rama de soluciones. La complejidad quedaría entonces $O(2^n) + O(n \cdot \log n) = O(2^n + (n \cdot \log n)) \in O(2^n \cdot n)$.

2.4. Programación Dinámica

Esta técnica se basa en utilizar la metodología de *divide and conquer* [4] y almacenar los resultados calculados para no tener que volver a hacerlo cuando se repita un problema. Para poder aplicar usar programación dinámica es necesario que se cumpla entonces:

- El problema pueda ser dividido en subproblemas y que la solución que se encuentre para cada uno de ellos sea óptima, para así construir la solución del problema inicial de forma óptima [5]
- Los subproblemas compartan soluciones[6], si no no tendría sentido almacenar las soluciones.

En este gráfico se puede ver cómo los nodos que ya se calcularon están en azul y los rojos serían nodos que no se calculan porque ya se obtuvo la información que le corresponde.



Dentro de este tipo de programación hay varios algoritmos de implementación. Uno es el de top-down (por el significado en ingles de de arriba para abajo) o memoización y el otro bottom-up (por el significado en ingles de abajo hacia arriba), aunque existen bibliografías que consideran sólo a top-down como programación dinámica.
En este análisis se implementarán de ambas formas de forma que se pueda ahondar más acerca de los beneficios de ambos.

2.4.1. Análisis

La idea de un algoritmo que implemente *divide and conquer* y almacenamiento de subproblemas, sugiere empezar pensando este problema y dividiéndolo en problemas pequeños.

Se puede afirmar que la solución, si existe, puede dividirse en dos casos: la solución no contiene al primer elemento de C y entonces la solución resulte de formar V con la mínima cantidad de elementos de C':

$$C' = C - \{e_1\}$$

Y en particular debe ser óptima. Supongamos por el absurdo que no. La solución al problema sería K $\leq$ |C'|. Supongamos que existe un Q <K que es solución. Pero esto no puede pasar, porque por la propia definición de nuestra solución, se va a devolver la mínima cantidad de elementos, entonces no puede existir un valor menor de elementos que sea solución.

La otra posible solución sería que e_1 sí sea parte de la solución, es decir, habría que encontrar elementos que sumen V' con los elementos del conjunto C' :

$$C' = C - \{e_1\}$$

$$V' = V - e_1$$

O sea que la solución óptima sería 1 + K, K $\leq$ |C'|, K solución. Si existiese un Q <1+K y la suma de los elementos de Q = V', es decir hay una solución mejor para el problema con C' y V'. Pero entonces ahora K = Q y la solución óptima es K + 1, donde el 1 representaría el primer e_1 que por el análisis de este caso formaba parte de la solución óptima.

[4] Metodología que busca simplificar un problema grande o complicado en problemas más chicos y simples de una forma recursiva
[5] Este concepto suele denominarse como: *principio del óptimo*
[6] Se denomina a esto subproblemas superpuestos y un ejemplo muy conocido es la sucesión de Fibonacci

2.4.2. Formulación recursiva

Se define PD una función que recibe un conjunto S, de elementos enteros no negativos, un valor V entero no negativo y un entero que representa el tamaño de S inicial, devuelve el mínimo cardinal del subconjunto s perteneciente a S que todos los elementos suman V; si no existe se devuelve n+1, n tamaño de S, simbolizando la imposibilidad de encontrar solución.

$$PD(S,V,n) = \begin{cases} n+1 & \text{si } |S| = 0 \wedge V \neq 0 \\ 0 & \text{si V} = 0 \\ PD(\{e_2, e_3, \ldots, e_n\}, V, n) & \text{si } e_1 > V \\ 1 & \text{si } e_1 = V \\ \text{mín}(1 + PD(\{e_2, e_3, \ldots, e_n\}, V - e_1, n), PD(\{e_2, e_3, \ldots, e_n\}, V,)) & \text{otro caso} \end{cases}$$

Cabe resaltar que a modo de una formulación más intuitiva, cada llamado se hace envíando un S modificado, algo que en una implementación tendría un costo muy grande de copias, es por eso que en código se utilizará una referencia (nunca se copia) al conjunto inicial (y no se modificará) y para saber los límites del conjunto en el llamado recursivo actual se usa un flag *desde* que hace referencia a los elementos que se pueden usar en ese llamado.

2.4.3. Algoritmo Top Down

```
AUX-TOPDOWN(C, V)
 1  if |C| = 0 then
 2          if V = 0 then
 3                  sol ← 0
 4                  Devolver 0
 5          else
 6                  sol ← −1
 7                  Devolver −1
 8  elseif V = 0 then
 9          sol ← −1
10          Devolver −1
11  soluciones ← matriz de n X V con −1 en los valores
12  soluciones_{0,0} ← 0
13  sol ← Topdown(C, V, 0, soluciones, |C|)
14  if sol > |C| or sol = −1then
15          Devolver −1
16  else
17          Devolversol
```

```
TOPDOWN(C, V, desde, memoria, n)
 1  if desde < 0 then
 2          if V = 0 then
 3                  devolver : 0
 4          else
 5                  devolver : n + 1
 6  if NoExiste(C, V, desde, memoria) then
 7          if c_desde == V then
 8                  Guardar(1, V, desde, memoria, n)
 9          else
10                  if C_desde > V then
11                          valor ← Topdown(C, V, desde − 1, memoria, n)
12                          Guardar(valor, V, desde, memoria, n)
13                  else
14                          valor1 ← 1 + Topdown(C, V − C_desde, desde − 1, memoria, n)
15                          valor2 ← Topdown(C, V, desde − 1, memoria, n)
16                          valor3 ← min(valor1, valor2)
17                          Guardar(valor3, V, desde, memoria, n)
18  Devolver : −1 si memoria_{desde,V} ≥ n
19  Devolver : memoria_{desde,V} si no
```

Tenemos dos algoritmos, el primero que es el que se encarga de analizar si el problema no es un caso base y devuelve la solución que corresponde y en caso contrario inicializa lo necesario para poder aplicar el algoritmo

de Top-Down. Las cosas que se inicializan son, la memoria, implementada sobre una matriz de n x (V+1)[7], donde se va a reutilizar los subproblemas ya calculados, el flag desde que permite a Top-Down saber desde cuál es el problema actual que hay que empezar a dividir en subproblemas.

El algoritmo de Top-Down primero preguntar si ya se calculo previamente el resultado del problema y en caso contrario lo calcula y lo almacena en memoria, que se utiliza mediante referencia para no hacer copias y tenerla todo el tiempo actualizad .

Es importante recalcar que tanto Aux-Top-Down como Top-Down utilizan a C por referencia, para no estar haciendo copias, que cuando n es grande podrían significar tiempos de ejecución mucho mayores.

La explicación de complejidad es un poco más difícil que para el próximo algoritmo porque en este hay llamados recursivos que hay que demostrar que no hacen cosas imprevisibles y arruinan la complejidad.

Imaginemos a la matriz de n x V, y pensemos que todavía no se calculó ningún resultado. Empezamos de la última fila i y columna j. Puede haber uno o dos llamados recursivos. Si hay uno, necesariamente la posición del proximo subproblema es la de (i-1)(j). Si hay dos, una de esas posiciones será (i-1)(j) y la otra (i-1)(j-C_k), donde k es, en este caso 1, pero que va de 1 a n. La cuestión es que más allá de que se llaman al mismo tiempo, hay un orden. Ese orden permite que cuando se hacen los dos llamados, primero se va a resolver uno y después el otro y si se llegara a dar que el llamado dos cae en una posición de un subproblema que ya calculó el otro llamado, es trivial que la solución se consigue en O(1) en ese llamado y cualquiera que utilice problemas ya calculados por el primer llamado. Si esto lo extendemos a toda la matriz, después del primer o los dos primeros llamados, uno de ellos se va terminar de resolver y recién ahí se resolverá el otro. Y en particular este comportamiento se repite para todos los llamados que generen los llamados del primer llamado (digamos $L1$). Como la matriz es finita, para cada fila (n) $L1$ va a generar dos llamados como máximo. Cada llamado recursivo se genera porque no había solución para el subproblema. Como la cantidad de subproblemas es n x V y los llamados recursivos se resuelven en orden, la máxima cantidad de llamados tiene que ser n x V, porque si no estaríamos diciendo que se van a hacer llamados recursivos para problemas ya calculados y esto no puede ser porque después de calcularlos los guardamos y particularmente al trabajar por referencia, al actualizar la memoria en un determinado llamado, todos los llamados anteriores que todavía no fueron resueltos tienen la memoria actualizada.

2.4.4. Algoritmo Bottom Up

```
BottomUp(C, V, n)
 1  memoria ← matriz de n · V de enteros vacia
 2  cargarConValores(memoria, n + 1)
 3  memoria_{0,0} ← 0
 4  for i ← 2 to n
 5  for j ← 1 to V
 6  if C_i > j then
 7          memoria_{i,j} ← memoria_{i−1,j}
 8  else
 9          valor1 ← memoria_{i−1,j}
10          valor2 ← 1 + memoria_{i−1,j−c_j}
11          memoria_{i,j} ← min(valor1, valor2)
12  Devolver : −1 si memoria_{n,V} ≥ n
13  Devolver : memoria_{n,V} si no
```

Inicialmente se inicializa la estructura donde vamos a almacenar los resultados de todos los subproblemas. Particularmente se inicializa la primera fila (la 0) con el valor n+1, que simbolizaría la imposibilidad de formar el valor objetivo con 0 cosas, excepto en la posición 0,0, que la definimos como 0.

Esta implementación se corresponde de forma muy evidente con su nombre: si imaginamos un problema como la raíz de un árbol y a los hijos como sus subproblemas es muy intuitivo el funcionamiento. Este algoritmo se basa en empezar resolviendo los subproblemas más chicos y resolver el problema que los generó, es decir, su padre. Dado que el padre de todos es el problema inicial, si repetimos este procedimiento mientras el padre tenga algún predecesor, terminaríamos resolviendo el problema inicial.

Como consecuencia de este funcionamiento tan estructurado, se resuelven todos los posibles subproblemas y no los necesarios, como Top-Down, de modo que es esperable que si el tamaño de C o el valor objetivo son grandes, se notará en la performance. Por otro lado la justificación de la complejidad es más simple, dado que no hay llamados recursivos que analizar, sino que se recorren las n x V celdas y se guarda el valor que le corresponde, que viene de alguna celda correspondiente a la fila anterior y alguna columna. Como la primera fila de la matriz tiene los çasos base", cada uno de las n x V iteraciones es O(1) porque simplemente pone en la celda actual el valor de otra celda o el valor del mínimo de otras celdas, que accede en O(1). Entonces es trivial que la complejidad es O(n x V).

[7] Se le suma 1 simplemente para tener una columna que se corresponda con el valor objetivo, dado que los indices empiezan en 0 y van hasta tope-1. No modifica la complejidad asintótica esto

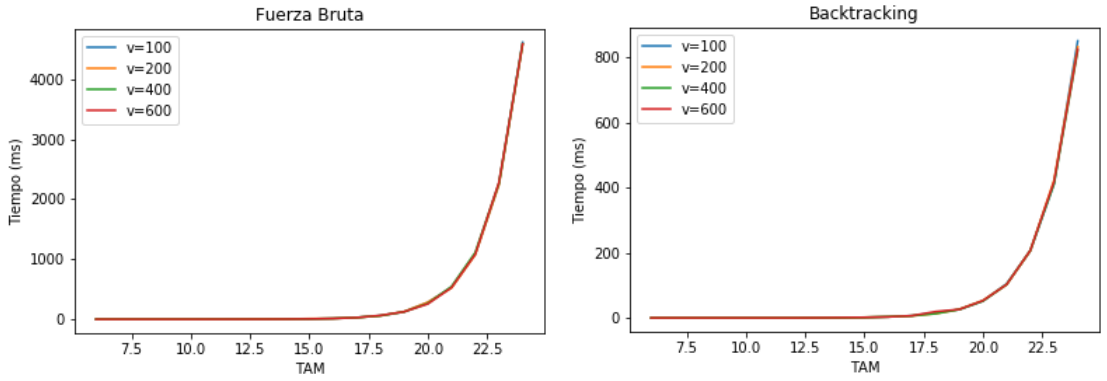
3. Experimentación

Para la experimentación primero se probará con casos de test para comprobar la correctitud de los algoritmos. Sin embargo y dado que instancias excesivamente grandes demandarían mucho tiempo en hacer el cálculo a mano para comprobar los resultados, se aprovechará la etapa de experimentación con instancias generadas en Python no solo para guardar la información necesaria para la experimentación sino que también para cada instancia se comparará cada uno de los resultados de los cuatro algoritmos. Evidentemente no es una prueba exacta pero da mayor seguridad a la hora de la correctitud.

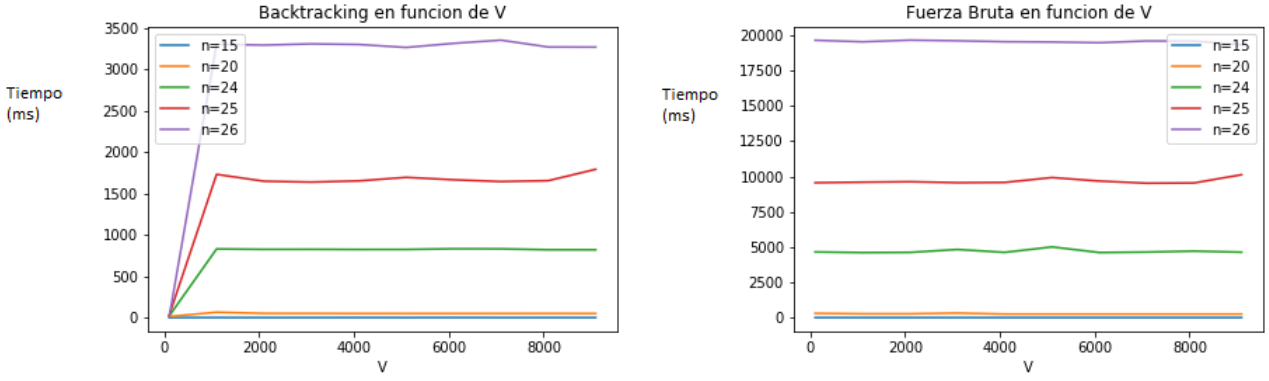
Las pruebas que se hagan con valores random se harán utilizando la libreria numpy en Python y para el análisis de los resultados de los algoritmos se usará: pandas, seaborn, matplotlib.pyplot y jupyter notebook. Además, se testearán 4 veces cada instancia en cada algoritmo para sacar un promedio de los tiempos de ejecución reduciendo al mínimo los posibles ruidos. Para la ejecución de los algoritmos y sus tiempos se utilizará una porción de código mostrada en una clase de laboratorio de experimentación.

3.1. Fuerza Bruta y Backtracking: exponenciales

Los primeros análisis sería interesante empezarlos comparando los algoritmos más básicos como fuerza bruta y backtracking. Posiblemente el algoritmo de fuerza bruta no permita ser computado para tamaños de C excesivamente grandes y los otros algoritmos sí, entonces sería razonable que se experimenten los algoritmos de forma cruzada y estableciendo relaciones y no todos de forma simultánea porque podría perderse información. Además, la implementación de los algoritmos sugiere que ante contextos de peor caso y |C| razonablemente chico, mientras que para casos de C grandes serían algoritmos sin uso práctico.

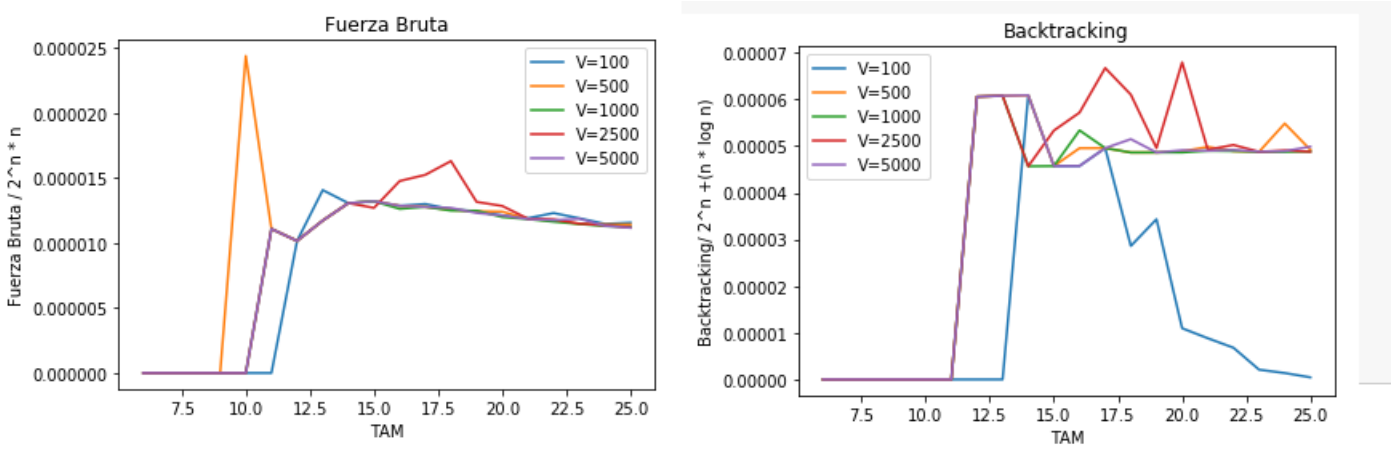


Podemos ver que ambos algoritmos tienen pinta de tener la complejidad exponencial respecto de n anteriormente analizada. Y también se puede inferir que el valor objetivo no es una variable para la complejidad de estos algoritmos, ya que vemos que con diferentes valores objetivos parecen ser iguales crecimientos asintóticos.



Con estos gráficos podemos confirmar que el valor objetivo no es un factor dentro de la complejidad de estos algoritmos. Podemos ver como a medida que el valor objetivo crece, para n variados en general, los algoritmos son asintóticamente constantes. En Backtracking podemos ver que en los primeros valores de V esto no se cumple, pero es porque en esos valores de V la relación con los elementos de C permite efectuar podas, mientras que para V excesivamente grandes, donde ya no hay podas, el valor de V no afecta. Como conclusión, Fuerza Bruta no es afectada por el valor objetivo, mientras que Backtracking podría tener casos donde la relación de

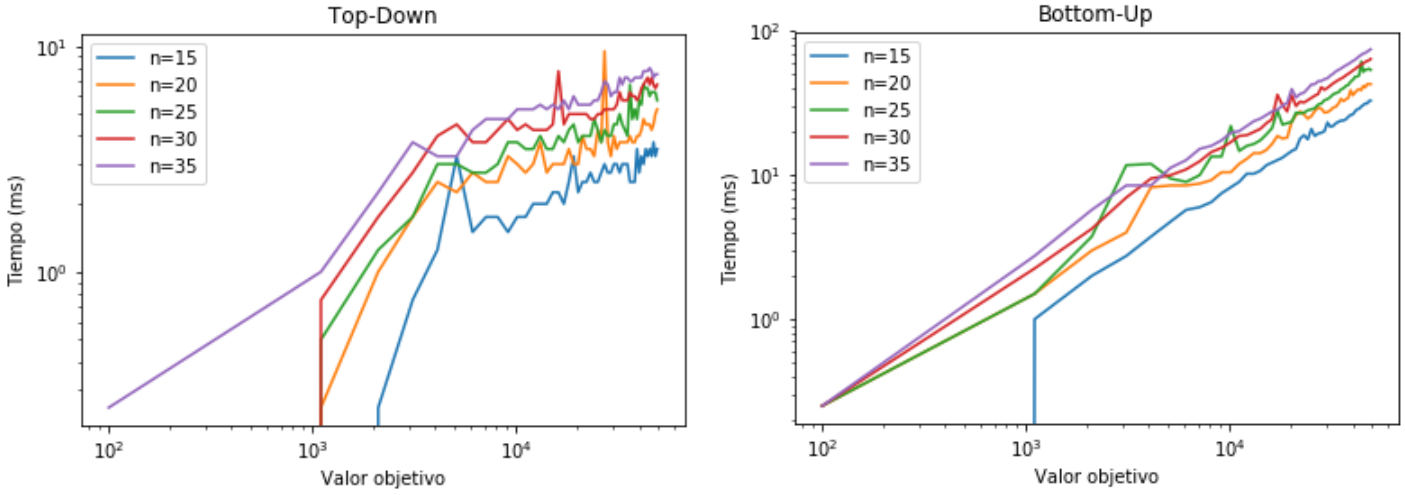
los elementos de C y el valor objetivo permite encontrar una o más soluciones y esto genera podas que en la práctica reduce la complejidad teórica.



Finalmente podemos terminar de afirmar que los algoritmos cumplen con la complejidad teórica, que son efectivamente exponenciales en el peor caso y que las variaciones del valor objetivo no tienen una correlación con la complejidad. El único comentario es que se puede ver como para V = 100 sigue manteniéndose que no constante respecto a la exponencial del backtracking, y eso se debe a como vimos en los anteriores casos, en esos V empieza a encontrar soluciones y funcionar algunas podas y por ello la complejidad es menor. Para los demás casos en donde V es de mucho mayor orden que los elementos de C, backtracking es exponencial (particularmente a la función $2^n \cdot n \log n$ y fuerza bruta a $2^n \cdot n$).

3.2. Top Down y Bottom Up: lineales en n x V

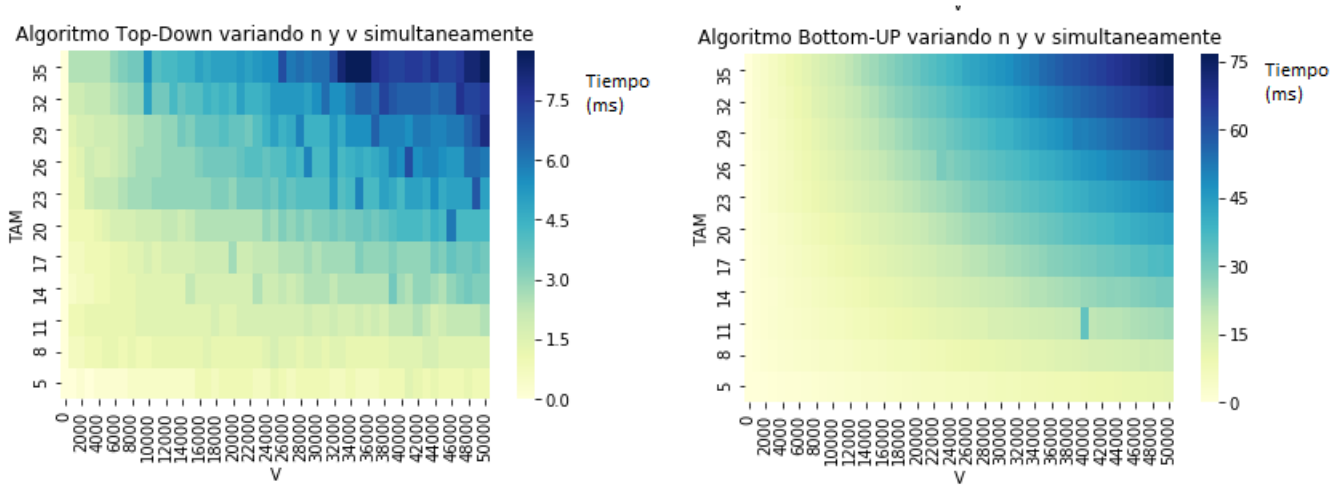
Como antes, comenzaremos analizando con los peores casos y tratando de llegar a ver que cumplen la complejidad anteriormente analizada. En Bottom-Up vimos que siempre resuelve todos los subproblemas en una matriz de n x V, y en Top-Down también existe una matriz de n x V (complejidades espaciales en teoría asintóticamente iguales en ese punto), pero a diferencia sólo resuelve los que necesita, de modo que esperamos ver ciertas diferencias en los mapas de calor.



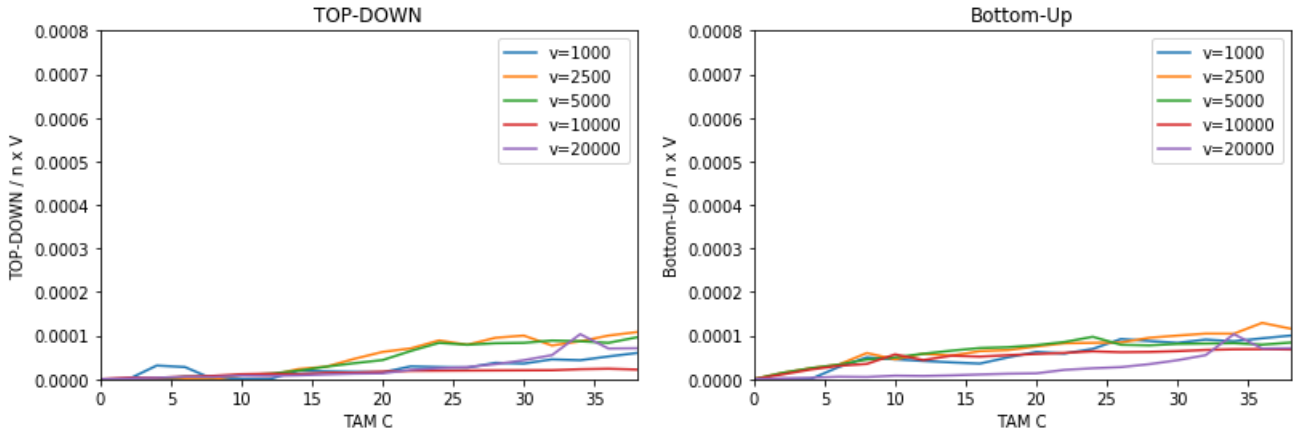
Los gráficos están en escala logarítmica en ambos ejes para una mejor visualización, sobre todo en el eje x donde el valor objetivo va desde 100 a 50000.

En ambos podemos inferir que existe un crecimiento dependiente de n y v , porque si bien los algoritmos crecen a medida que aumenta V, también se ve que para C cada vez más grandes el crecimiento también existe. Por otro lado, hay que remarcar que el Top-Down tiene no solo un crecimiento más aplastado, producto de no calcular todos los subproblemas, sino menores tiempos.

Veamos los algoritmos en algo que nos permita comparar de forma más clara las variaciones en función de V y n.



Con los mapas de calor se puede observar cómo los aumentos de n y V influyen directo en la performance de los algoritmos de programación dinámica. Sin embargo, también hay que notar que en Top-Down el proceso de aumentar n y V no empeora la performance del algoritmo de forma tan escalonada como en Bottom Up sino que el tipo de instancias definen en parte la performance que tendrá, en función de los subproblemas que tenga que resolver.



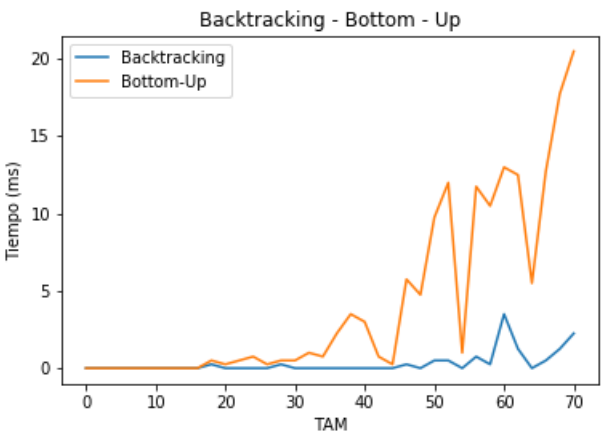
De nuevo, con este último podemos terminar de asegurar que los algoritmos tienen la complejidad anteriormente analizada ($n \cdot V$).

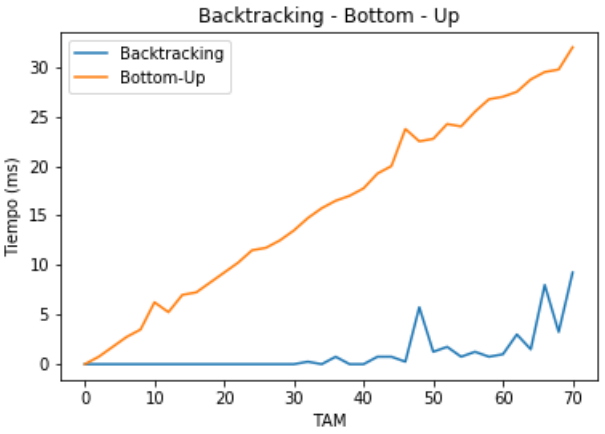
3.3. Backtracking vs programación dinámica

Nota: dado que tanto Top Down y Bottom Up son algoritmos de programación dinámica, vamos a utilizar para los análisis sólo al último como representante de esta metodología.

Ahora trataremos de buscar contextos en cada uno de estas metodologías se impongan.

Empecemos analizando los mejores casos para Backtracking. Estos necesitan ser instancias en donde se apliquen podas la mayor cantidad de veces posibles, y esto pasa si todo el tiempo se pueden encontrar mejores soluciones o evitamos grandes ramas de soluciones porque los valores de los elementos de C a partir de una posición son mayor que V (y aca ayuda que el Backtracking ordena a C). Fijemos para esto valores de V entre 0 y n^2 y los valores de C entre 0 y V, de modo de tener ciertas relaciones entre los valores y que a su vez sean lo suficientemente abarcativos.

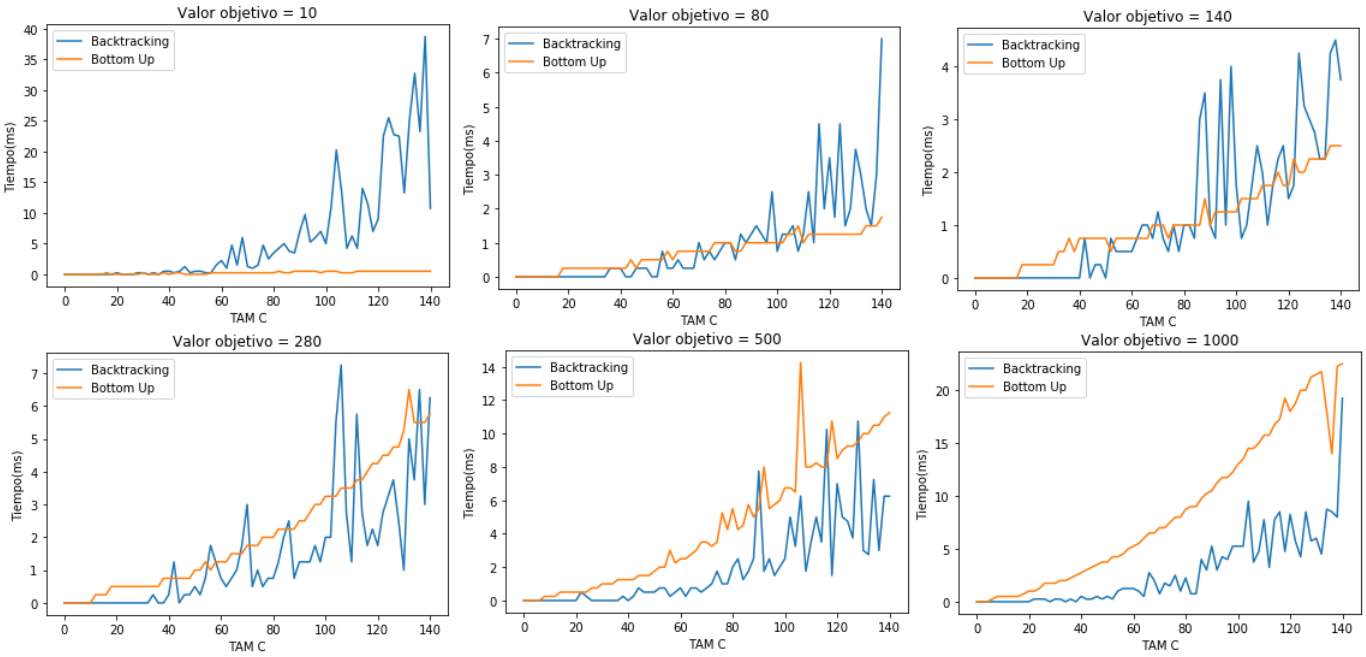




Es apreciable que en este tipo de instancias, la performance de un algoritmo de programación dinámica tiende a ser menor a la de Backtracking, que obtiene muchos beneficios de sus podas. Lo más interesantes es ver (en el segundo gráfico) que si fijamos a V = 10000 (un numero considerablemente grande) y los elementos de C siguen entre 0 y V, el algoritmo de programación dinámica es aún menos eficiente, y es razonable por lo antes analizado, que vimos que depende fuertemente de n y V.

Entonces.. cuál serían los casos en donde programación dinámica sea mejor que Backtracking? Bueno por un lado ya sabemos que cualquier instancia en donde los elementos de C son de mucho menor orden que V, Bactracking se comporta exponencialmente, como un algoritmo de fuerza bruta. En esos casos ya sabemos que lo mejor es programación dinámica. Pero existirán otros casos donde Backtracking se beneficie por las podas y aún así los algoritmos de programación dinámica son mejores?

Es evidente que necesitamos instancias donde n x V no sea demasiado grande, y donde las podas de Bactracking funcionen, porque ya sabemos que sino es exponencial y es trivial que algún algoritmo de programación dinámica es mejor. Analicemos casos en donde n sea suficientemente representativo, y analicemos fijando diferentes V chicos y los elementos de C en orden de V. Esperamos que V chicos y elementos de C en ese orden favorezca las podas de Backtracking y que el tamaño grande de C y los elementos que estén "alejados" de V lo penalicen y ahí sacar la ventaja con los algoritmos de programación dinámica.



Es interesante ver la influencia del crecimiento de V para que la diferencia de performance empiece a igualarse y posteriormente a invertirse. Todavía más interesate si se mira con los anteriores gráficos, que se ve exactamente lo opuesto porque queríamos ver lo opuesto y que los dos grupos pueden verse como uno solo, en donde las diferentes etapas por las cuales los algoritmos se benefician o perjudican se hacen muy evidentes.

4. Conclusiones

Durante la presentación de este trabajo se presentaron diferentes técnicas algorítmicas para hallar las soluciones del conocido problema *suma de subconjuntos*. Cada una de estás técnicas responden a diferentes formas de abordar el problema para llegar a una solución exacta y particularmente en nuestro caso la mejor. Inicialmente se hizo un análisis de la literatura del problema, mencionando diferentes algoritmos y sus objetivos principales. Posteriormente se escribieron cuatro diferentes algoritmos y desde ese momento se plantearon interrogantes acerca de qué tipo de situaciones o contextos podrían beneficiarlos. Finalmente se realizaron experimentos que permitieron conocer un poco más en profundidad los puntos débiles y fuertes de las soluciones, así como también adquirir la información suficiente a la hora de tomar decisiones de cuál de ellos experimentar en función del contexto, los recursos disponibles o las necesidades.

Los algoritmos que utilicen fuerza bruta quedan relegados para situaciones muy singulares, con tamaños de C muy pequeños, y su ventaja responde a la facilidad con la que se puede diseñar. Por el lado de Backtracking se puede encontraron situaciones en donde sería interesante aplicarlo. Estas se pueden englobar en las que los elementos de C son muy grandes respecto de V, y si V llegara a ser grande incluso vimos que le ganaría a los algoritmos de programación dinámica.

Desde otro punto de las metodologías encontramos a Top-Down y Bottom up. Top Down fue un poco más rápido y también como era previsible, la performance de Bottom Up era directamente proporcional a los tamaños de n y V, y cuando estos eran muy grandes Top Down también lo padecía pero, como se vio en los mapas de calor, para muchas instancias al resolver sólo los subproblemas necesarios mejoraba mucho la performance respecto de Bottom Up. Este tipo de algoritmos son los mejores para resolver este problema cuando no se cumplen ciertas requerimientos como para aplicar Backtracking, porque no hay que olvidar que si C llegara a ser grande, con que una parte chica de los elementos no cumpla lo necesario, Backtracking ya es un algoritmo exponencial. Es decir, para utilizar esta técnica es necesario tener muy bien definido cómo serán V, |C| y los elementos de C para usarla.

5. Referencias

[3]Dynamic training subset selection for supervised learning in Genetic Programming, Chris GathercolePeter Ross.

[4]Generalizing Cryptosystems Based on the Subset Sum Problem, Aniket Kate and Ian Goldberg

[5]Thomas Cormen, Algorithms and Optimization, Approximation Algorithms, pág 1106

[6]G. J. Woeginger, Exact Algorithms for NP-Hard Problems: A Survey, Lecture Notesin Computer Science 2570, pp. 185-207

[7]Computing partitions with applications to the knapsack problem, Journal of the Association for Computing Machinery

[8]Donald Knuth. *The Art Of Computer Programming : Fundamentals Algorithms*

6. Cambios

1. Agregadas algunas cosas en la introducción.
2. La sección de Fuerza Bruta permaneció sin cambios.
3. Toda la sección de Backtracking fue modificada. Particularmente pseudocódigo, código fuente, justificación de complejidad, acorde a los cambios necesarios para llegar a la pedida.
4. Se sacó el código fuente y se agrandaron los gráficos.
5. Agregados experimentos con escala logarítmica.
6. Modificadas algunas explicaciones de programación dinámica de complejidad. En Top-Down particularmente se modificó pseudocódigo y código fuente.
7. Modificados y agregados gran parte de los experimentos, para rehacerlos por las modificaciones de algunos algoritmos y en otros casos para agregar casos.
8. Agregados títulos ilustrativos en los experimentos.
9. Modificada las conclusiones a partir de los nuevos experimentos.
10. Se hace la entrega con las instancias que se usaron para testear y para experimentar.