DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1: Jambo Tubo

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Semeria, Camilo	818/97	csemeria@dc.uba.ar
Mansini, Leo	318/19	lmansini@dc.uba.ar

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
https://exactas.uba.ar

1. Introducción

El *problema de jambotubo.*

De acuerdo a la consigna propuesta, una cinta transporta una secuencia de objetos hacia un tubo y debemos programar un autómata que decida por cada objeto si debe o no ser depositado en dicho tubo. Para tomar esa decisión, el autómata debe evaluar todos los objetos presentes en la lista a fin de encontrar la combinación que maximice la cantidad de objetos que se introducen en el tubo teniendo en cuenta que cada objeto posee un peso y que el tubo posee una resistencia que indica el peso máximo que tolera. A su vez, cada objeto posee una resistencia propia; dado que el tubo apila objetos, ésta resistencia indica el peso total que un objeto soporta sobre si, es decir, limita el paso máximo de la suma de todos los objetos que sean introducidos luego. Abstrayéndonos a un planteo con enfoque matemático-lógico, cada objeto puede verse como un número p_i (peso) y otro r_i (resistencia) y nos interesa maximizar la cantidad de sumandos de forma tal que cada suma posterior a un elemento j incluido en la misma sea como máximo su resistencia r_j, considerándose el propio tubo como un pseudo objeto cero de peso irrelevante y resistencia r_0, el cual no es contabilizado a la hora de dar el resultado, siempre teniendo además en cuenta que el orden de los sumandos no puede ser alterado siendo la única decisión posible considerarlo o ignorarlo. A fines prácticos asumiremos que ningún objeto puede tener peso cero. Fundamenta esta decisión que deseamos que le sea posible indicar que un objeto no resiste que se le ponga nada encima mediante asignarle resistencia cero, por lo que por muy despreciable que fuera el peso de cualquier objeto ulterior, asignarle cero indicaría que pueden incluirse ilimitado número del mismo y eso eventualmente causaría que la resistencia cero sea vencida y el modelado del problema falle. Por ello entendemos que es "menos grave" que haya que asignar un peso mínimo a un elemento toda vez que alcanzaría con hacer un ajuste de la granularidad del pesaje para no sufrir efectos en la búsqueda de un resultado óptimo, siendo por lo tanto un inconveniente mucho más sencillo de subsanar. De la misma manera asumimos que los pesos y las resistencias son números enteros. Esto último no causa pérdida alguna de generalidad toda vez que la unidad de medida de peso del sistema no está especificada ni restringida por lo que puede ser arbitrariamente pequeña si así se desea (gramos, microgramos, etc).

Ejemplo de una instancia del problema:
Tubo de resistencia $r_0 = 70$
Elementos $O_1...O_n$ (en formato $p_1|\ r_1$,...,$p_n|\ r_n$):
10|20, 15|50, 5|30, 15|30, 15|30, 15|30
Arrojara por resultado 4 debido a que si incluimos el O_1 su resistencia $r_1 = 20$ nos permitiría agregar $O3(p_3 = 5)$ y uno y sólo uno cualquiera entre $O2$, $O4$, $O5$ y $O6$. Mientras que si ignoramos $O1$ podremos agregar $O2$ y tres objetos cualesquiera entre $O3...O6$ pero no 4 porque la suma de $p_4...p_6 = 45 > r_3 = 30$. Dado que si excluimos O_2 sólo restan 4 objetos, se puede garantizar que no es posible un resultado mejor que 4 sin necesidad de cálculos adicionales.

A lo largo del presente trabajo estudiaremos distintos enfoques para resolver el problema y compararemos resultados: usaremos fuerza bruta, backtracking con poda por optimalidad y factibilidad y programación dinámica, para luego analizar las métricas obtenidas a partir del análisis de casos de prueba y comparar resultados empíricos con los órdenes teóricos.

2. Fuerza Bruta

Para aplicar fuerza bruta usaremos un enfoque recursivo. Entenderemos que en su estado inicial el sistema tiene una "resistencia actual" equivalente a la resistencia del tubo (r_0) y que a la hora de evaluar si cada elemento subsiguiente k ($1 \leq k \leq n$) debe agregarse o no, se pueden generar dos posibles caminos, o bien el elemento se agrega o bien no se agrega. De esta forma la cinta puede visualizarse como un número binario de n dígitos y garantizarse que se exploran todas las posibilidades. A fin de evaluar si una combinación obtenida resuelve o no el problema, la llamada recursiva mantendrá la resistencia actual si el elemento actual no se agrega o pasará el mínimo entre, la resistencia del objeto agregado y, la "resistencia actual" del sistema antes de considerarlo menos el peso del objeto actual, todas las combinaciones que tras evaluar el n-ésimo elemento resulten en "resistencia actual" negativa, no son soluciones del sistema, esto indica que o bien el tubo no resistió o un elemento fue aplastado. De las restantes, la que mayor cantidad de elementos haya incluido (que no necesita ser única) será la respuesta al problema.
Complejidad: $\Theta(2^n)$ siendo n la cantidad de elementos en la cinta.
Justificación: El caso base, cuando se analiza el ultimo elemento de la cinta, tiene complejidad constante. Todos los demás casos hacen 2 llamados recursivos, lo que da un sistema recurrente del tipo $T(n) = 2T(n-1) + O(1)$ ya que en los llamados se pasa a ver si se agrega o no el siguiente elemento, por lo que n disminuye en 1.
Los vectores se pasan por referencia para reducir tiempo de ejecución en cada llamado.

Algorithm 1 Algoritmo de Fuerza Bruta para JamboTubo.

```
1: function FB(int resistenciaJT, int n, vector<int> pesos, vector<int> soportes)
2:     return FBAux(resistenciaJT, cantElementos, pesos, soportes, 0)
3:
4: function FBAux(int resistenciaActual, int n, vector<int> pesos, vector<int> soportes, int
   nroElemento)
5:     if nroElemento == (n − 1) then                                          ▷ Caso base
6:         if r < 0 then res = −∞                                                  ▷ Θ(1)
7:         else
8:             if pesos[nroElemento] > r then res ← 0                              ▷ Θ(1)
9:             else
10:                res ← 1                                                         ▷ Θ(1)
11:    else                                                                    ▷ Caso recursivo
12:        res ← max(
13:        1 + FBAux(min(resistenciaActual−pesos[nroElemento], soportes[nroElemento]),
14:                n, pesos, soportes, nroElemento + 1),
15:                                                                    ▷ Agrego el elemento
16:        FBAux(resistenciaActual, n, pesos, soportes, nroElemento + 1)
17:                        ▷ No lo agrego
18:
19:        )                                                                       ▷ 2T(n − 1)
20:    return res
21:
```

3. Backtracking

Para aplicar backtracking usaremos el mismo criterio recursivo aplicado a Fuerza Bruta con al variante de que al evaluar el elemento k-ésimo de la cinta verificaremos si el estado actual del sistema tiene solución posible y, de ser así, si la mejor solución posible puede mejorar la mejor obtenida hasta el momento a partir de evaluaciones ya realizadas. Si lo primero no se cumple haremos una poda por factibilidad, si no se cumple lo segundo haremos una por optimalidad. En ambos casos, podemos garantizar que el problema se resuelve correctamente si las podas son correctas toda vez que no se descartan combinaciones que sean candidatas a ser la respuesta. Sobre la correctitud de las podas se argumentará a continuación.

Poda por factibilidad Dado que el problema se resuelve evaluando elemento a elemento las consecuencias de agregarlo o no, podemos establecer sin lugar alguno a duda que si el resultado de agregar un elemento genera el aplastamiento de un objeto o supera la tolerancia del tubo (o sea, si supera la "resistencia actual" del sistema) no habrá combinación alguna de los restantes objetos que logre revertir ese resultado, toda vez que eso implicaría la necesidad de objetos con peso negativo. Por ello, podemos descartar evaluar las combinaciones restantes en caso de agregar ese objeto k-ésimo con la seguridad de no estar perdiendo ninguna respuesta factible (Poda por factibilidad 2: elemento aplastante, líneas de 19 a 22). De la misma forma, si a la hora de evaluar el k-ésimo objeto la resistencia actual del sistema es cero, podemos simplemente truncar toda evaluación ulterior a sabiendas de que la única forma de mantener la respuesta factible es no agregar más elementos, por lo que alcanza con retornar en $O(1)$ le respuesta al subproblema "cuántos elementos se pueden agregar a un tubo de resistencia 0" que es, por supuesto, 0 (Poda por factibilidad 1: tubo lleno, línea 10).

Poda por optimalidad Para evaluar optimalidad necesitamos ante todo tener conocimiento del mejor resultado evaluado hasta el momento. Para ello guardaremos ese resultado en una variable global, dado que de nada serviría tenerlo en las llamadas recursivas toda vez que, salvo cuando se poda una rama de evaluación, ninguna llamada recursiva cuenta aún con resultado alguno. De la misma forma, es conveniente tener disponible o bien el tamaño de la entrada original o bien de la actual, a fin de tener disponibilidad de los elementos restantes por evaluar. Hecho esto, agregamos como parámetro a la llamada recursiva la cantidad de elementos incluidos al momento y consideraremos que si la cantidad restante de elementos por evaluar más la de incluidos hasta el momento no es mayor que el óptimo ya encontrado, no hay forma de que esa rama de evaluación resulte en un mejor resultado (Poda por optimalidad 2: elementos restantes insuficientes, lineas de 34 a 36). De la misma forma, como los pesos se asumen enteros positivos (distintos de cero, para zanjar toda ambigüedad sobre criterio de positividad), si la resistencia actual del sistema hace imposible agregar la cantidad de elementos necesarios para superar el óptimo, podemos descartar la rama sin riesgo alguno de perder una respuesta que supere a la la obtenida. En otros términos, si el resultado de sumar los elementos ya agregados a la resistencia actual

del sistema no supera al óptimo, no queda nada más por evaluar por ese camino (Poda por optimalidad 1: resistencia insuficiente, lineas de 26 a 29).

Algorithm 2 Algoritmo de Backtracking para Jambotubo.

```
1:
2:  int mejorResultado ← 0
3:  function BT(int resistenciaJT, int n, vector<int> pesos, vector<int> soportes)
4:      return BTAux(resistenciaJT, cantElementos, pesos, soportes, 0, 0)
5:
6:  function BTAux(int resistenciaActual, int n, vector<int> pesos, vector<int> soportes, int nroElemento,
    int rtaAcutal)
7:
8:      elementosRestantes ← n − nroElemento − 1
9:      res ← 0                                                                          ▷ O(1)
10:     if (r == 0) then return 0          ▷ Poda por factibilidad 1: tubo lleno
                                                                                         ▷ O(1)
11:     if nroElemento == (n − 1) then                                                   ▷ Caso base
12:                                                                                      ▷ O(1)
13:         if r < 0 then res = −∞
14:         else
15:             if pesos[nroElemento] > r then res ← 0
16:             else res ← 1
17:                 rtaActual++
18:     else                                                                             ▷ Caso recursivo
19:         if pesos[nroElemento] > r then          ▷ Poda por factibilidad 2: elemento aplastante
20:             res ← BTAux(resistenciaActual, n, pesos, soportes nroElemento+1, rtaActual)
21:                                                                                      ▷ T(n − 1)
22:         else
23:             resistenciaAgregandome ← min(r-pesos[nroElemento], soportes[nroElemento])
24:
25:             //Poda por optimalidad 1: resistencia insuficiente
26:             if resistenciaAgregandome ≥ 0 AND
27:                     ((rtaActual + 1 + resistenciaAgregandome) ≤ mejorResultado) then
28:                 agregando ← 0                                                       ▷ O(1)
29:             else
30:                 //caso recursivo agregando
31:                 agregando ← 1+BTAux(resistenciaAgregandome, n, pesos, soportes,
32:                                     nroElemento + 1, rtaActual + 1)                 ▷ T(n − 1)
33:             //Poda por optimalidad 2: elementos restantes insuficientes
34:             if mejorResultado > (rtaActual + elementosRestantes - 1) then
35:                 sinAgregar ← 0                                                      ▷ O(1)
36:             else                                                 ▷ caso recursivo sin agregar
37:                 sinAgregar ← BTAux(resistenciaActual, n, pesos, soportes,
38:                                    nroElemento+1, rtaActual)                        ▷ T(n − 1)
39:             res ← max(agregando, sinAgregar)          ▷ O(1) o T(n − 1) o 2T(n − 1)
40:     if res ≥ 0 AND rtaActual > mejorResultado then
41:         mejorResultado ← rtaActual                                                  ▷ O(1)
42:     return res
```

Complejidad: $O(2^n)$, siendo n la cantidad de elementos en la cinta.
Justificación: El caso base, cuando se analiza el ultimo elemento de la cinta, es O(1). Si no se aplica ninguna poda, la ejecución es idéntica a la de Fuerza Bruta, pero si se aplican, se salva el llamado recursivo a uno o dos de los dos casos: agregar o no el elemento al Jambotubo.
Suponiendo un caso donde en cada llamado se aplica alguna poda, salvando uno de los dos llamados recursivos, se tiene un tiempo de ejecución $T(n) = T(n-1) + O(1)$ teniendo como resultado una complejidad lineal. Naturalmente, si en algún llamado las podas no realizan ninguno de los dos llamados, el tiempo de ejecución es $O(1)$. Por otro lado el peor caso es el de que no se aplique ninguna poda, esto se comporta igual que Fuerza Bruta, $O(2^n)$.
Los vectores se pasan por referencia para reducir tiempo de ejecución en cada llamado.

4. Programación Dinámica

Tiene sentido usar *programación dinámica* cuando nos encontramos con un problema que tiene superposición de subproblemas. en ese caso lo que hacemos es plantear una solución recursiva al problema principal y guardar las soluciones de las subinstancias que vayamos teniendo que resolver, de forma tal que si una llamada se reitera, al tener el valor precalculado, nos evitamos tener que volver a calcular toda una subrama del problema. Para éste caso utilizaremos la siguiente función recursiva:

$$f(i,R) = \begin{cases} 0 & \text{si } i = n \wedge p_i > R, \\ 1 & \text{si } i = n \wedge p_i \leq R, \\ f(i+1,R) & \text{si } i < n \wedge RA(i,R) < 0, \\ \text{máx}\{1 + f(i+1, \text{mín}\{r_i, RA(i,R)\}), f(i+1,R)\} & \text{si } i < n \wedge RA(i,R) \geq 0 \end{cases} \quad (1)$$

Donde:

- n es la cantidad de elementos en la cinta
- i es el elemento de la cinta que se está evaluando ($0 < i \leq n$)
- r_0 es la resistencia inicial del JamboTubo
- r_i es la resistencia del elemento i
- p_i es el peso del elemento i
- RA(i, R) es la resistencia actual tras restar R - p_i
- R es la resistencia actual del tubo y sus elementos al momento de realizar la llamada, la cual equivale a la resistencia del tubo cuando éste está vacío, o la menor de las resistencias del tubo y cada objeto restadas a las mismas los pesos de todos los objetos que tienen sobre si. Nótese que siempre $0 \leq R \leq r_0$.

Correctitud

1. si $i = n$ estamos ante un subproblema de un solo elemento por lo que la solución es trivial: si $p_i \leq R$ el elemento puede ser agregado al tubo y la respuesta es 1, caso contrario no puede ser agregado y la respuesta es 0.

2. Si $i < n$ no estamos en el último elemento del subproblema, si además resulta que RA(i,R) < 0, o sea, que si se agrega el actual elemento el tubo colapsa, sólo nos queda buscar el mejor resultado posible a partir de i+1 excluyendo el elemento i.

3. Si $i < n$ pero RA(i,R) ≥ 0, no estamos en el último elemento y a su vez el actual puede o no agregarse al tubo, por lo que tenemos que buscar la mejor respuesta entre agregar el elemento i o no, teniendo en cuenta que si lo agregamos la resistencia actual disminuye a RA(i,R), es decir, al mínimo entre r_i y R - p_i.

Memoización Como se aprecia en la función recursiva eq.1 cuando evaluamos el elemento k-ésimo el resultado no varía de acuerdo a cuantos elementos anteriores se han agregado. Eso reduce las 2^n posibles evaluaciones del árbol de soluciones del problema (número que surge de lo ya explicado en la sección 2: fuerza bruta) a posibles i x R instancias donde $0 \leq i \leq n$ y $0 \leq R \leq r_0$ siendo r_0 la resistencia inicial del JamboTubo. De esa forma, utilizando como estructura de memoización auxiliar una matriz de $(r_0 + 1)$ x n (no consideramos el caso $i = n$ en la matriz por trivialidad) podemos garantizar que cualquier evaluación de un paso del problema posterior a la primera no requiere evaluar la subrama correspondiente sino que se responde en $O(1)$.
Complejidad: $O(n * r)$ siendo n la cantidad de elementos en la cinta y r la resistencia del jambotubo.
Justificación: Para justificar el orden conviene analizar el algoritmo de la siguiente manera: Las llamadas recursivas 15 y 25 están precedidas por consultas a la estructura de memoización que verifican que no se ha hecho esa misma llamada con anterioridad. A su vez, como se ve en la línea 33, cada llamada a la función deriva en una escritura en nuestra estructura de memoización en una coordenada que, por lo antedicho, sabemos que estaba indefinida. De ello se desprende que la cantidad total de llamadas recursivas a nuestra función está acotada por las dimensiones de la matriz de memoización. Finalmente, debido a que el único costo no constante del algoritmo es el resultante de las llamadas recursivas y que la cantidad de consultas a la estructura de memoización por cada iteración no es dependiente de n, se verifica que el orden temporal de $O(nxR)$ siendo n la cantidad de elementos de la cinta y R la resistencia inicial r_0 del JamboTubo.

Algorithm 3 Algoritmo de Programación Dinámica para Jambotubo.

```
1: vector<vector<int>> M
2: M[i][j] ← UNDEFINED ∀ i, j 0 ≤ i ≤ r, 0 ≤ j < n
3: function PD(int resistenciaJT, int n, vector<int> pesos, vector<int> soportes)
4:     return PDAux(resistenciaJT, cantElementos, pesos, soportes, 0)
5:
6: function PDAux(int resistenciaActual, int n, vector<int> pesos, vector<int> soportes, int
   nroElemento)
7:
8:     res ← 0                                                                  ▷ O(1)
9:     if nroElemento == (n − 1) then                                           ▷ O(1)
10:        if pesos[nroElemento] > resistenciaActual then res ← 0
11:        else
12:            res ← 1
13:    else
14:        if M[resistenciaActual][nroElemento + 1] == UNDEFINED then
15:            sinAgregar ← PDAux(resistenciaActual, n, pesos, soportes, nroElemento + 1)
16:                                                                  ▷ T(r, n − 1)
17:        else
18:            sinAgregar ← M[resistenciaActual][nroElemento + 1]               ▷ O(1)
19:        resistenciaAgregandome ← min(r-pesos[nroElemento], soportes[nroElemento])
20:
21:        //si la resistencia agregando el actual elemento genera aplastamiento,
22:        descarto la opción
23:        if resistenciaAgregandome ≤ 0 then
24:            if M[resistenciaAgregandome][nroElemento + 1] == UNDEFINED then
25:                agregando ← 1 + PDAux(resistenciaAgregandome, n, pesos, soportes,
26:                                      nroElemento + 1)
27:                                                  ▷ T(resistenciaAgregandome, n − 1)
28:            else
29:                agregando ← 1 + M[resistenciaAgregandome][nroElemento + 1]   ▷ O(1)
30:        else
31:            agregando ← 0
32:        res ← max(agregando, sinAgregar)
33:    M[resistenciaActual][nroElemento] ← res
34:    return res
```

5. Experimentación

En esta Sección están presentados los experimentos sobre el tiempo de ejecución de los algoritmos descritos en las secciones anteriores. Los experimentos se realizaron en una workstation con microprocesador Intel Core i7-10700F @ 2.9 GHz y 32 GB de memoria RAM.

Se aclara que toda ejecución de los algoritmos fue repetida 25 veces, y el valor de tiempo de ejecución que se tomó para el análisis fue la mediana de todos ellos.

5.1. Métodos

Las configuraciones y métodos utilizados durante la experimentación son los siguientes:

- **FB**: Algoritmo 1 de Fuerza Bruta de la Sección 2.
- **BT**: Algoritmo 2 de Backtracking de la Sección 3.
- **BT-F**: Algoritmo 2 con excepción de las líneas de 26 a 29 y de 34 a 36 , es decir, solamente aplicando podas por factibilidad.
- **BT-O**: Similar al método **BT-F** pero solamente aplicando podas por optimalidad, o sea, descartando las líneas 10 y de 19 a 22 del Algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 4.

5.2. Instancias

Para los experimentos realizados se crearon 7 familias de instancias a fin de que los algoritmos sean evaluados en un conjunto de casos variado. Esto es porque, si bien el tiempo de ejecución de los algoritmos tienen una dependencia directa con el *tamaño* de la entrada, un análisis de los métodos utilizados indica que hay instancias "mejores" o "peores" en tiempo de ejecución para cada algoritmo. Esto significa que la complejidad temporal de los algoritmos depende, además, de ciertos aspectos que pueden tener o no los elementos de entrada. Por esta razón resulta interesante probar distintas familias de datasets para los distintos algoritmos desarrollados.

Se describen los 7 datasets a continuación.

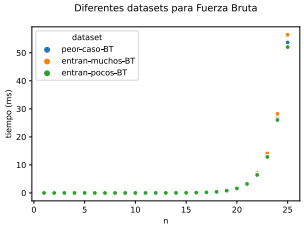
1. **entran-muchos-BT**: Instancias en las que entran muchos elementos en el Jambotubo. Casos razonablemente buenos para backtracking con poda de optimalidad y malos para backtracking con poda de factibilidad. Los parámetros son aleatorios pero lo esperado en probabilidad es que entre por lo menos el 75 %.
   $1 \leq n \leq 50$, $9n \leq r \leq 10n$, $9n \leq soporte_i \leq 10n$, $\frac{r}{n} \leq peso_i \leq \frac{r}{0{,}75n}$.
2. **entran-pocos-BT**: Instancias en las que entran pocos elementos en el Jambotubo. Casos razonablemente buenos para backtracking con poda de factibilidad y malos para backtracking con poda de optimalidad. Los parámetros son aleatorios pero lo esperado en probabilidad es que entre alrededor del 10 %.
   $1 \leq n \leq 50$, $9n \leq r \leq 10n$, $9n \leq soporte_i \leq 10n$, $\frac{r}{n0{,}1} \leq peso_i \leq 1{,}1\frac{r}{n*0{,}1}$.
3. **mejor-caso-BT-O**: Instancias en las que entran todos los elementos en el Jambotubo. Mejor caso para backtracking con poda de optimalidad dado que el mejor resultado en $O(n)$ y luego se poda el resto.
   $1 \leq n \leq 100$, $r = 10n$, $soporte_i = 10n$, $peso_i = 10$. Por lo que $\sum_{i=0}^{n} peso_i = r$.
4. **mejor-caso-BT-F**: Instancias en las que no entra ningún elemento en el Jambotubo. Mejor caso para backtracking con poda de factibilidad porque se podan todos los casos donde se agrega algún elemento.
   $1 \leq n \leq 100$, $r = 10n$, $soporte_i = 10n$, $peso_i = r + 1$. De modo que $min\{peso_i\} > r$.
5. **peor-caso-BT**: Instancias en las que entra aproximadamente la mitad de los elementos en el Jambotubo. Ya que ninguna de las dos podas es particularmente buena, este es el peor caso para el algoritmo de backtracking que utiliza ambas. Los parámetros son aleatorios pero lo esperado en probabilidad es que entre alrededor del 40 %.
   $1 \leq n \leq 50$, $9n \leq r \leq 10n$, $9n \leq soporte_i \leq 10n$, $\frac{2r}{n} \leq peso_i \leq \frac{3r}{n}$.
6. **mejor-caso-DP**: En estas instancias se varía tanto la resistencia como la cantidad de elementos en la cinta dado que la complejidad de programación dinámica depende de ambas. Es el mejor caso porque todos los pesos son iguales, lo cual maximiza la superposición de casos.
   $1000 \leq n \leq 8000$, $1000 \leq r \leq 8000$, $soporte_i = resTubo$, $peso_i = 1\ \forall i$.
7. **peor-caso-DP**: Los pesos en esta familia de instancias son las potencias de 2. La razón por la que optamos por esa distribución es porque la superposición de problemas cuenta con que al llegar a un elemento en la cinta, halla más de una manera de haber conseguido la resistencia actual. Si los pesos son las potencias de 2, (o de cualquier número), podemos pensarlos como una base, por lo que si tomamos dos subconjuntos *distintos* del conjunto de pesos, y sumamos sus elementos, la suma será diferente. Por una cuestión de representación de enteros, el programa funciona bien hasta que se llega a números del orden de 2^{25}, por lo que lo fijamos como máximo de los pesos. Esto provoca una superposición de problemas superior a la mínima si $n > 25$, pero sigue siendo el peor caso realizable.
   $1 \leq n \leq 100$, $9n \leq resTubo \leq 10n$, $9n \leq soporte_i \leq 10n$, $peso_i = 2^{k \bmod 25}\ \forall i, 0 \leq k \leq n$.

5.3. Experimento 1: Complejidad de Fuerza Bruta

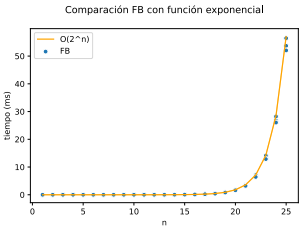
En este experimento analizamos el rendimiento del algoritmo Fuerza Bruta en 3 datasets diferentes, **entran-pocos-BT**, **entran-muchos-BT** y **peor-caso-BT**. Nuestra hipótesis, justificada en la Sección 2, es que para cualquier instancia de tamaño n, Fuerza Bruta tiene el mismo rendimiento, $\Theta(2^n)$. Por eso, elegimos los datasets de casos más variados posibles.

La Figura 1a demuestra empíricamente que Fuerza Bruta se comporta igual para los 3 datasets, sugiriendo que el tiempo de ejecución crece exponencialmente con n. En la Figura 1b se puede ver que una función exponencial $f(n) = c2^n, c \in \mathbb{R}$ explica muy bien la evolución del tiempo de ejecución del algoritmo, y esta fuerte relación es demostrada en la Figura 1c, obteniéndose además un coeficiente de correlación de Pearson de $\approx 0{,}999347$.

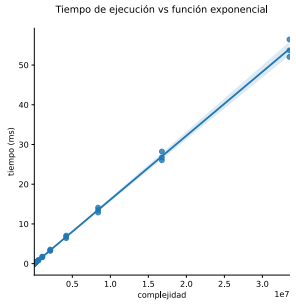
Hemos verificado la hipótesis de que Fuerza Bruta es un algoritmo con complejidad $\Theta(2^n)$, y que además esa complejidad no depende de la naturaleza de la instancia en la que se ejecuta.



(a) Tiempo de ejecución en los datasets, **entran-pocos-BT**, **entran-muchos-BT** y **peor-caso-BT**.



(b) Tiempo de ejecución contra complejidad esperada (2^n).



(c) Tiempo de ejecución vs función exponencial en n

Figura 1: Análisis de complejidad del método FB.

5.4. Experimento 2: Complejidad de Backtracking

En esta experimentación correremos el algoritmo de Backtracking con ambas podas en sus mejores y peores casos. Los datasets serán entonces **mejor-caso-BT-O**, **mejor-caso-BT-F**, **peor-caso-BT**.

Los mejores casos de Backtracking son donde sus podas son lo más efectivas posible. Para ello, pensamos en cada poda por separado, por lo que hay mejor caso para la poda de optimalidad, y mejor caso para la poda de factibilidad.

En primer lugar el mejor caso de la poda de factibilidad es el caso en el que ningún elemento entra en el jambotubo por ser muy pesado. En este caso la poda provoca que se haga recursión solo en el caso de "no agregar", y por ende el algoritmo es $O(n)$.

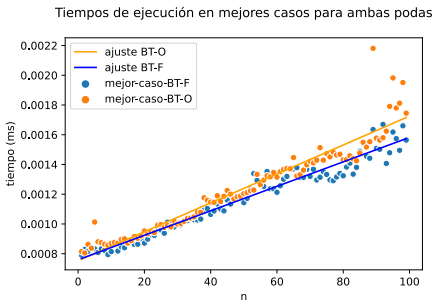
En segundo lugar la poda de optimalidad tiene máxima eficiencia cuando entran todos. Esto se debe a que en la recursión se realiza antes el caso de "agregar", y eso provoca que se calcule el mejor resultado (que sera igual a n) antes de que se pruebe de no agregar el primer elemento. Cuando se evalúe si es posible superar el mejor resultado sin agregar el primer elemento, en ese momento se poda, y como calcular el resultado de agregar todos es $O(n)$, esa es la complejidad de este mejor caso.

En tercer lugar, los peores casos son donde entra la mitad de los elementos, esto es porque ese es el punto en el cual la acción de ambas podas se minimiza, por lo que se maximiza el tiempo de ejecución.

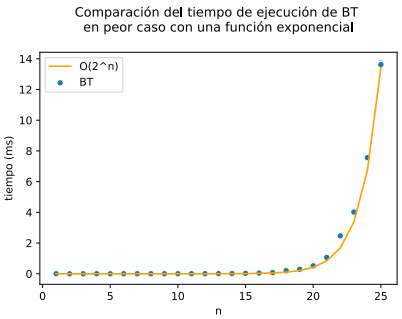
Nuestras hipótesis, entonces, son que Backtracking tendrá complejidad lineal en los mejores casos, y complejidad $O(2^n)$ en los peores casos, ya que en muchas veces hará dos llamados recursivos, como lo hace siempre Fuerza Bruta.

En la Figura 2a se ve claramente que en los mejores casos Backtracking tiene complejidad lineal. Esta relación lineal es analizada en las Figuras 2c y 2d, obteniéndose un coeficiente de Pearson de $\approx 0{,}976478$ para el buen caso de factibilidad y de $\approx 0{,}954257$ para el buen caso de optimalidad.

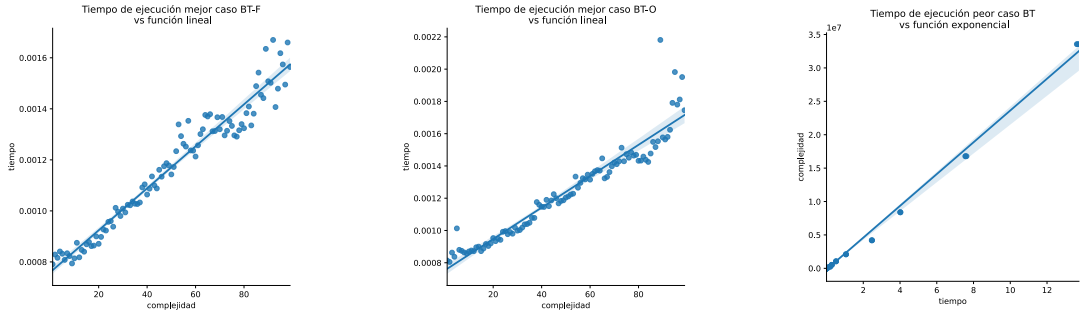
La evolución del tiempo de ejecución en función de n parece crecer exponencialmente en los peores casos de Backtracking, como se muestra en la Figura 2b. Luego se tiene un gráfico del tiempo de ejecución vs una función exponencial en n (Figura 2e), la relación lineal entre ambas es clara y el coeficiente de Pearson es de $\approx 0{,}99757$.



(a) Tiempo de ejecución de mejores casos de **BT**



(b) Tiempo de ejecución de peores casos de **BT**.



(c) Correlación entre el tiempo de ejecución para mejores casos por factibilidad y función lineal en n.

(d) Correlación entre el tiempo de ejecución para mejores casos por optimalidad y función lineal en n.

(e) Correlación entre el tiempo de ejecución para peores casos y función exponencial en n.

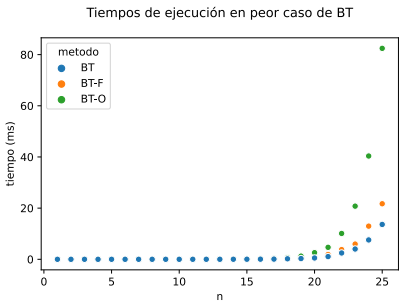
Figura 2: Análisis de complejidad del método **BT** para el data set bt-peor-caso.

5.5. Experimento 3: Efectividad de las podas

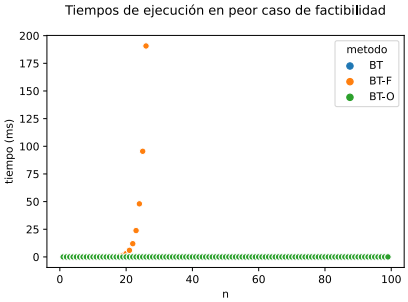
Para este experimento analizamos como se comportan los algoritmos de Backtracking que aplican solo una poda.

Primero comparamos la versión que aplica ambas podas con las versiones que aplican una de las dos, ejecutando esas 3 versiones, **BT, BT-F, BT-O**, en el dataset **peor-caso-BT**. Los resultados en la Figura 3a muestran que ambas podas contribuyen al mejor rendimiento posible de Backtracking, dado que la versión con las dos podas es la más eficiente.

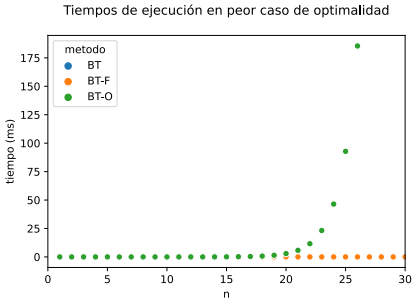
Luego nos interesa comparar cuanto tardan ambas podas por separado en el mejor caso de optimalidad, y en el mejor caso de factibilidad, es decir los datasets **mejor-caso-BT-O** y **mejor-caso-BT-F**. Aquí lo que esperamos es ver que el mejor caso de optimalidad no es nada bueno para la poda de factibilidad, de hecho esperamos que se comporte como fuerza bruta, que no pode nunca. Esto es porque al entrar todos los elementos al jambotubo, ninguno es podado por ser demasiado pesado. Lo mismo esperamos para el caso inverso, en el mejor caso de factibilidad, no entra ningún elemento y por lo tanto no se consigue ningún resultado que actualice el valor de mejor resultado para poder podar soluciones parciales por optimalidad, por lo que **BT-O** en el dataset **mejor-caso-BT-F** tendrá complejidad exponencial. La hipótesis es confirmada en las Figuras 3b y 3c, donde en su peor caso, cada poda tiene un crecimiento exponencial. Como en esas figuras no es tan claro el rendimiento de las podas más eficientes, se graficaron las dos podas en sus mejores casos por si solas en las Figuras 3d y 3e los cuales son naturalmente iguales en tiempo de ejecución a las Figuras 2c y 2d.



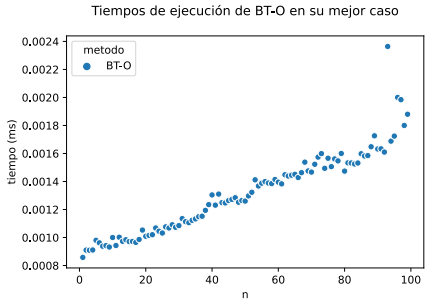
(a) Efectividad de las podas para peores casos.



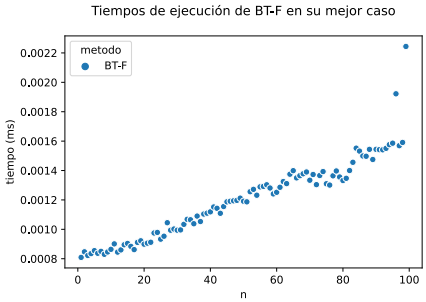
(b) Comparación de cada poda en el dataset **mejor-caso-BT-O**.



(c) Comparación de cada poda en el dataset **mejor-caso-BT-F**.



(d) Tiempo de ejecución de **BT-O** en su mejor caso.



(e) Tiempo de ejecución de **BT-F** en su mejor caso.

Figura 3: Comparación de efectividad en las podas.

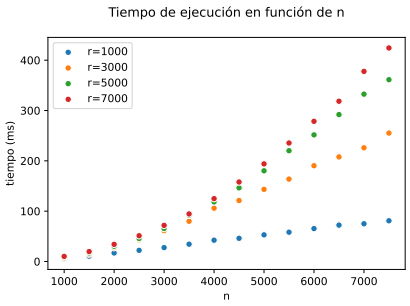
5.6. Experimento 4: Complejidad de Programación Dinámica

Como en **DP** la complejidad depende de dos parámetros, para este experimento nos enfocamos primero en analizar los tiempos de ejecución en función de n, para 4 valores de r, y después ver los tiempos de ejecución en función de r para 4 valores de n. Además vemos si la complejidad de $O(n * r)$ es correcta al ejecutarlo en los datasets elegidos, **dp-mejor-caso**, y **dp-peor-caso**.

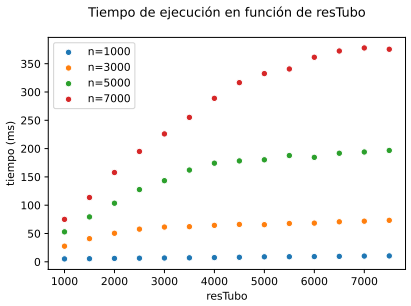
Las figuras 4a y 4b sugieren que el tiempo de ejecución crece tanto con n como con r. Por ejemplo, los tiempos de ejecución para $r = 1000$ son menores que los tiempos de ejecución para $r = 7000$, y lo mismo pasa cuando se fijan valores en n.

Hay que tener en cuenta que el algoritmo cuenta con algo similar a una poda por factibilidad. La estructura de memoización no guarda soluciones parciales invalidas ($r < 0$), por lo que instancias en donde no entran todos los elementos, como en todos los casos donde $n > r$ (todos los pesos valen 1), se resolverán mas rápido que instancias en donde sí entran, es decir los casos $n \leq r$.

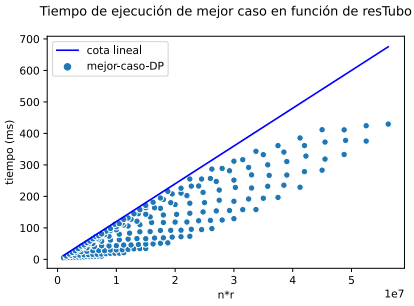
Para verificar si los tiempos de ejecución siguen una relación lineal con el producto entre n y r, se crearon las Figuras 4c y 4d. En la Figura 4c se puede establecer una función lineal en $n*r$ como cota, ya que los tiempos son iguales o menores a esta función. En cambio, al ser los peores casos, los tiempos en la Figura 4d están mucho más pegados a esta cota, siguen una relación lineal con $n * r$. Esto verifica la complejidad temporal del algoritmo de $O(n * r)$.



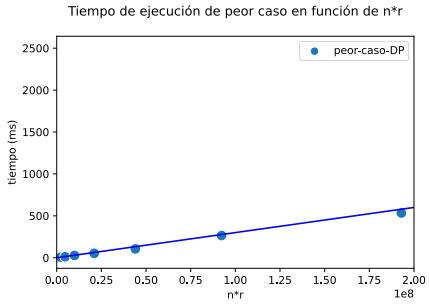
(a) Tiempo de ejecución en función de n, para $r = 1000$, 3000, 5000 y 7000.



(b) Tiempo de ejecución en función de r, para $n = 1000$, 3000, 5000 y 7000.



(c) Tiempo de ejecución de **DP** en su peor caso.



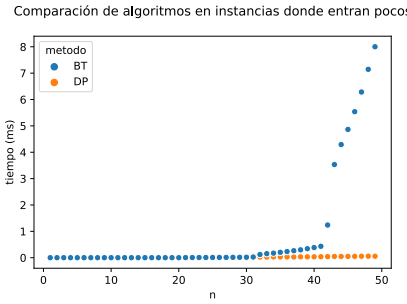
(d) Tiempo de ejecución de **DP** en su mejor caso.

Figura 4: Análisis de buenos y malos casos para el algoritmo de Programación Dinámica

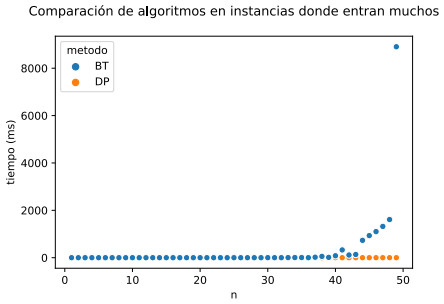
5.7. Experimento 5: Backtracking vs Programación Dinámica

Como último experimento comparamos dos algoritmos diferentes, Backtracking y Programación Dinámica.

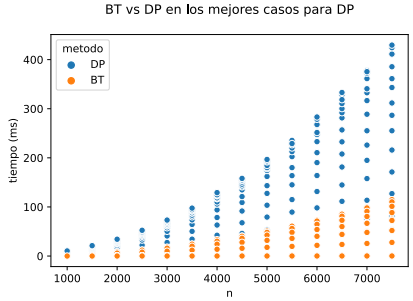
Los datasets que consideramos más "promedio", y que no son ni el peor ni el mejor caso de ninguno de los algoritmos, son **entran-muchos-BT** y **entran-pocos-BT**. En las Figuras 5a y 5b están los tiempos de ejecución de ambos algoritmos en función de n, en donde se ve una mucha mayor eficiencia de **DP**, dado que **BT** empieza a crecer exponencialmente. Sin embargo, es notable que si comparamos tiempos en los mejores y peores casos de **DP**, en los dos **BT** es mejor, como se puede observar en las Figuras 5c y 5d. Es decir, en los casos promedio el más eficiente es **DP**, pero en donde **DP** es más eficiente que en ningún otro caso, es superado por **BT**. Esto sucede porque en los mejores casos de **DP**, o entran todos, o entran todos hasta un punto donde la resistencia es demasiado pequeña, es decir son casos en donde se aplican las podas de **BT** muy eficientemente. En los peores casos de **DP**, simplemente entran todos los elementos, por lo que la poda de optimalidad es, valga la redundancia, óptima.



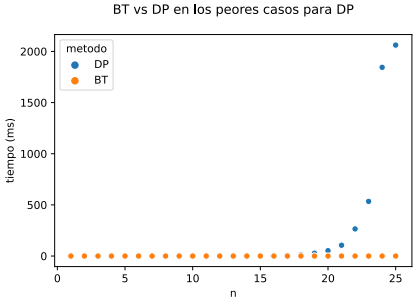
(a) Comparación de tiempos de ejecución en el dataset **entran-pocos-BT**.



(b) Comparación de tiempos de ejecución en el dataset **entran-muchos-BT**.



(c) Comparación de tiempos de ejecución en el dataset **mejor-caso-DP**.



(d) Comparación de tiempos de ejecución en el dataset **peor-caso-DP**.

Figura 5: Comparación de tiempos de ejecución entre Backtracking y Programación Dinámica.

6. Conclusiones

Se verificaron complejidades temporales para los tres algoritmos, **FB** $\in \Theta(2^n)$, **BT** $\in O(2^n)$ y **PD** $\in O(n*r)$

Al comparar los tres algoritmos utilizados para resolver el problema nos encontramos con que, cómo era de esperarse, Fuerza Bruta es el menos eficiente. Backtracking nos ofrece una mejora notable en performance, sobre todo en casos en que la respuesta al problema es cercana a n o a 0 siendo el más eficiente para manejar esos casos particulares, mientras que programación dinámica representa una mejora significativa en la mayoría de las entradas posibles.

Se aprecia también que al probar construir casos muy malos para programación dinámica se logra que su performance decaiga pero también se observa que la naturaleza de estos casos (pesos siendo potencias de una misma base) lo convierte en un ejemplo a priori poco probable.

Finalmente, es importante notar que a pesar de que programación dinámica utiliza mucha memoria, sobre todo cuando la resistencia del tubo es un R grande, esto es optimizable dado que en muchos casos a causa de la granularidad de las medidas de resistencia y peso muchas posiciones de la matriz no se utilizan (ejemplos con valores múltiplos de 100) y aún cuando la optimización no resulte tan trivial se puede recurrir a técnicas de mapeo que eviten el uso de filas innecesarias en la matriz.