DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP1: Optimizando Jambo-tubos

17 de mayo de 2020 — Algoritmos y Estructuras de datos III

Integrante	LU	Correo electrónico
Oshiro, Javier Esteban	715/09	javieroshiro@hotmail.com
Castro, Jonatan Daniel	63/18	jonatan.dan@hotmail.com

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

Índice

1. Introducción

En el presente trabajo práctico se pide realizar la programación de robots pertenecientes a la empresa Jambo, con el fin de optimizar una de las diversas operaciones en sus locales. Particularmente, se pide programar un robot que realizará el servicio de empaque en Jambo-tubos. El servicio consiste en un robot que decidirá cuales de los productos que le llegan por una cinta transportadora serán empacados en un tubo, con el objetivo de lograr apilar la mayor cantidad de productos sin que se rompan. Cada producto llega por la cinta en un orden determinado, y se conoce su peso y resistencia individual, por lo que el robot debe manejar esta información para no romper el tubo ni los productos apilados dentro del mismo por el propio peso de cada uno [1].

Por ejemplo, supongamos que tenemos 3 elementos que se transportan en la cinta en el orden presentado, tienen pesos individuales $\{1, 2, 3\}$ y resistencias individuales $\{3, 2, 1\}$. La resistencia del Jambo-tubo será de 4, la solución es 2 y los conjuntos que las representan son los elementos $\{1, 2\}$ o $\{2, 3\}$, en ambos casos el peso acumulado no supera los 4 del Jambo-tubos, y ninguno de los elementos se rompe al agregar el siguiente.

En el informe desarrollamos tres técnicas de programación que nos permiten abordar el problema de los Jambo-tubos: Fuerza Bruta(FB), Backtracking(BT), y por último Programación Dinámica(PD).

2. Fuerza Bruta

El algoritmo de Fuerza Bruta recorre todas las posibles soluciones sin importar que sean factibles o no. En nuestro caso, se tienen en cuenta todas las posibles formas de ingresar los productos en el tubo, y luego se determina cuál es la cantidad máxima de productos que se pueden ingresar sin romper ninguno de los productos apilados. Podemos observar que en todos los casos se revisan la totalidad del conjunto de soluciones para después determinar la solución óptima por lo que no podemos distinguir mejor y peor caso.

2.1. Implementación y correctitud

El algoritmo implementado es recursivo y en cada paso se revisan dos posibilidades sobre el producto actual en cinta, se agrega o no al tubo. Se genera el árbol de recursión con todos los casos posibles agregando un elemento por vez, o sea se desarrollan todas las ramas completas siendo sus hojas las posibles soluciones. Finalmente basta con seleccionar la que cumple que maximiza la cantidad de productos ingresados al tubo sin romper el tubo ni los productos apilados, esto junto con el árbol de recursión completo nos garantiza la correctitud del algoritmo.

Para saber si se rompe algún producto dentro del Jambo-tubo guardamos en la variable $minR$ la mínima resistencia restante entre los productos 1, para ello calculamos en cada paso el mínimo entre el nuevo producto agregado y la diferencia entre la resistencia mínima restante y el peso del nuevo producto. De esta forma siempre estamos verificando que el producto de menor resistencia restante no se rompa, asegurando el estado del resto de los productos de mayor resistencia. Si se rompe el elemento de menor resistencia restante luego ya sabemos que ese caso no es solución. Al tener en cuenta en $minR$ la resistencia del Jambo-tubo al inicializar el algoritmo con $minR = R$, estamos cubriendo también el caso en que este se rompe.

Los parámetros que recibe la función 1 son:

- i: índice de producto en la cinta.
- W: peso acumulado de los productos en el tubo.
- k: cantidad máxima de productos ingresados.
- $minR$: mínima resistencia restante entre los productos en el tubo.
- *aplastados*: indica si algún producto es aplastado por otro.

Algorithm 1 Algoritmo de Fuerza Bruta.

1: **function** $FB(i,\ W,\ k,\ minR,\ aplastados)$
2: **if** $i = n$ **then**
3: **if** $W <= R \wedge !aplastados$ **then return** k **else return** 0
4: $aplastadosAux \leftarrow aplastados\ ||\ (minR - w[i] < 0)$
5: **return** máx$\{FB(i+1, W, k, minR, aplastados), FB(i+1, W+w[i], k+1, min(minR - w[i], r[i]), aplastadosAux))\}$.

2.2. Complejidad

La complejidad del algoritmo es $\theta(2^n)$ para todos los casos, pues en cada paso recursivo genera dos llamados a la misma función, una para el caso en que se agrega el producto al tubo, y otra para el caso en que no. El árbol de recursión completo cuenta con $n + 1$ niveles contando la raíz, y siendo cada hoja una solución posible. La cantidad de nodos nos indica la cantidad de llamados recursivos realizados que son exactamente 2^n. El resto de las operaciones elementales y comparaciones se realizan en $\theta(1)$.

3. Backtracking

Similar al algoritmo de fuerza bruta, intenta recorrer el árbol de recursión con todas las posibles combinaciones pero evitando revisar algunas ramas según una implementación de podas que puede ser por factibilidad en caso de que esa rama no lleve a una solución factible, o por optimalidad en caso de que no lleve a una solución óptima. Por la propiedad dominó podemos asegurarnos que ninguna solución del sub-árbol generado por el nodo sobre el cual se aplica la poda llevará a una solución válida u óptima, puesto que en cada paso se agregan productos luego no es posible que una solución actual filtrada pase a ser válida en posteriores pasos.

3.1. Implementación y correctitud

En este caso la correctitud se garantiza de forma similar a la de FB ya que el algoritmo BT 2 sólo agrega sobre este último podas que finalizan la búsqueda de soluciones por la rama que está siendo evaluada. Estas podas se implementan como filtros sobre el algoritmo FB. Se agrega un nueva variable global *kOptimo* respecto a la versión de FB que mantiene el k que maximiza la cantidad de productos sin romper el tubo ni los elementos apilados, y se elimina el parámetro *aplastados* pues esa funcionalidad se implementa en la poda de factibilidad.

Poda por factibilidad Implementada sobre la idea de evitar continuar con ramas en que la suma de los pesos de los productos en el Jambo-tubo superen la resistencia total del mismo, o rompan alguno de los productos dentro del tubo.

Poda por optimalidad Implementada sobre la idea de evitar continuar por ramas en que sabemos que ingresando en el Jambo-tubo todos los elementos restantes, no llegaremos a superar el número de elementos ingresados óptimo.

3.2. Complejidad

La implementación es igual a la del algoritmo de FB pero en cada paso se verifican las dos condiciones para las podas, por lo que sigue teniendo peor caso con complejidad $O(2^n)$ sumado a la complejidad propia de cada poda. Sin embargo tiene un mejor caso con complejidad menor cercana a lineal $O(n)$, resultante de aplicar las podas y evitar recorrer todo el árbol de recursión. Teniendo podas que se resuelven con operaciones elementales y comparaciones en $\theta(1)$, no se agrega complejidad adicional.

Algorithm 2 Algoritmo de Backtracking.

```
1: function BT(i, W, k, minR)
2:     if i = n then
3:         if W <= R ∧ minR >= 0 then
4:             kOptimo ← max(kOptimo, k)
5:             return kOptimo
6:         else
7:             return 0
8:     if poda_factibilidad ∧ (W > R || minR < 0) then return 0
9:     if poda_optimalidad ∧ (kOptimo >= (k + n − i − 1)) then return 0
10:    return máx{BT(i + 1, W, k, minR), BT(i + 1, W + w[i], k + 1, min(minR − w[i], r[i]))))}.
```

4. Programación Dinámica

La programación dinámica tiene como característica evitar calcular valores que ya fueron computados en pasos anteriores. Por la naturaleza recursiva del problema, podemos tener casos en que un sub-árbol se calcule más de una única vez generando cómputo innecesario. En este método de programación, resolvemos este problema utilizando un diccionario de memoización que contenga todos los valores calculados hasta el momento para poder reutilizarlos.

4.1. Implementación y correctitud

La implementación es top-down, recursiva y similar a la de los métodos anteriores, con el agregado de que al momento de hacer el paso recursivo se verifica que los valores no se hayan computado ya. Si no se calculó, se lo procesa y se guarda en la matriz de memoización. Si se calculó anteriormente, no se entra al paso, sino que se obtiene el valor directamente de la matriz de memoización. Los casos base serían: cuando el peso actual aplasta un elemento de abajo o destruye el Jambo-tubo, o cuando se llega al final de la cinta transportadora.
La recursión es la misma que en Backtracking y Fuerza Bruta, de forma que esta versión del algoritmo también incurre en un árbol de recursión que garantiza que se evalúen todas las posibilidades (evitando recalcular). Como cada valor de la matriz es el máximo k que resuelve el problema hasta ese momento, luego basta con tomar el máximo de todos ellos.

Memoización: La estructura de memoización utilizada es una matriz que tiene por tamaño de fila la cantidad total de elementos de la cinta, y por tamaño de columna la resistencia del Jambo-tubo. Cada valor de la matriz corresponde con el máximo número de elementos ingresados en el Jambo-tubo en el paso i con la mínima resistencia resultante de haber metido o no elementos anteriores, es decir que contiene el k óptimo hasta ese momento. La memoización se da cuando en distintas ramas de la ejecución del algoritmo, en un paso i, se repite el valor de la resistencia acumulada. Un ejemplo de esto puede ser una instancia en donde el segundo elemento es igual en resistencia y peso al tercer elemento. Luego, van a haber una rama en donde no se agregue el segundo elemento y sí el tercero, y va a existir otra rama en donde se haga lo contrario. Como las ramas coinciden en resistencia acumulada va a entrar en juego la memoización evitando calcular valores ya conocidos.

4.2. Complejidad

La complejidad del algoritmo de programación dinámica está determinada por la cantidad de estados que se resuelven y el costo de resolver cada uno de ellos. Sabemos que la matriz tiene una dimensión de $n*R$ que es la cantidad de elementos por la resistencia total del tubo. Para inicializar la matriz insumimos $O(n*R)$ y en el algoritmo a lo sumo se resuelven $O(n*R)$ estados distintos que equivale a completar la matriz. Como las operaciones son elementales o calcular un máximo

Algorithm 3 Algoritmo de Programación Dinámica.

```
1: function PD(i, Ractual)
2:     if Ractual < 0 then return −1
3:     if i = n then return 0
4:     if m[i][W] == −1 then
5:         Rproxima = min(Ractual − w[i], r[i])
6:         m[i][W] ← max(DP(i + 1, Ractual), 1 + DP(i + 1, Rproxima))
7:     return m[i][W].
```

entonces cada cálculo se resuelve en $\theta(1)$. Finalmente el algoritmo tiene complejidad $O(n * R)$ en el peor caso.

5. Experimentación

Las experimentaciones tienen como objetivo verificar las complejidades y comparar entre si los diferentes algoritmos ya presentados. Los algoritmos fueron codificados en su totalidad en lenguaje C++, y todos los experimentos fueron ejecutados en una única PC con las siguientes características: procesador Intel Core i3-6100U a 2.3 Ghz, 3Mb L3 cache, y 6 GB de RAM DDR4.

5.1. Instancias

Para la experimentación realizamos experimentos sobre cinco datasets que representan tipos de instancia de diferentes características:

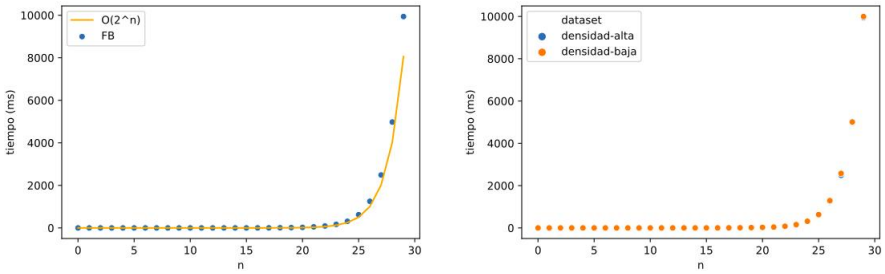
- densidad-alta: devuelven k alto, muchos elementos son considerados para ser ingresados en el tubo.
- densidad-baja: devuelven k bajo, pocos elementos son considerados para ser ingresados en el tubo.
- mejor-caso-BT: no entra ningún elemento en el tubo.
- peor-caso-BT: entran todos los elementos en el tubo.
- dinámica: instancias con n y R variable, y resistencias individuales aleatorias.

Es necesario hacer algunas aclaraciones: cuando escribamos BT-F, nos referimos al método de Backtracking por poda de factibilidad, y BT-O por poda de optimalidad. Y cuando usemos BT a secas, si no se indica lo contrario, estaremos usando ambas podas al mismo tiempo.
Además argumentamos que el dataset mejor-caso-BT es mejor caso sólo para BT-F, y lo mismo sucede con peor-caso-BT y BT-O. Ahora, mejor-caso-BT será un dataset de peor caso para BT-O, y peor-caso-BT será una instancia de peor caso para BT-F.

5.2. Experimento 1: Fuerza Bruta

En el primer experimento, resolvemos el problema utilizando el método de Fuerza Bruta ya introducido en la sección 1. Para probar la hipótesis de que el algoritmo tiene complejidad $\theta(2^n)$ graficamos en la figura 1a el tiempo tardado por el algoritmo según el n contra la función exponencial 2^n, con n la cantidad total de elementos de la instancia del problema. Vamos a utilizar un dataset de instancias del problema que tienen soluciones de valores k altos.

Como se puede ver en la figura 1a, el tiempo que tarda el algoritmo está solapado con la función exponencial, respaldando nuestra hipótesis de que efectivamente esa es la complejidad.
Para demostrar que el tipo de instancias no influye en el tiempo tardado, y que lo único que

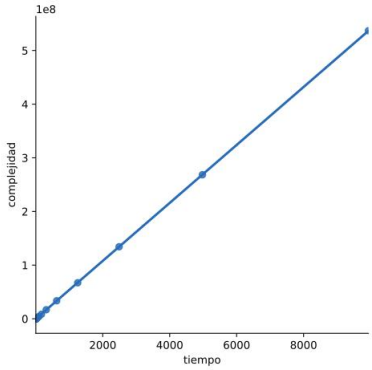


(a) Tiempo contra función exponencial (b) Densidad alta contra densidad baja

Figura 1: Análisis del método FB.

tiene peso es la cantidad de elementos total, ejecutamos el algoritmo contra un dataset distinto en calidad de tipos de instancias, pero con la misma cantidad de productos totales en la cinta. Este nuevo dataset tendrá instancias de *kOptimo* alto, en contraste con las de *kOptimo* bajo del ultimo dataset.

Podemos ver en la figura 1b que los gráficos se solapan. Esto indica que ambos presentan comportamiento exponencial, lo cual explica que el algoritmo de Fuerza Bruta no discrimina entre tipos de instancias, sino que solo depende de la cantidad total de elementos que vienen por la cinta.



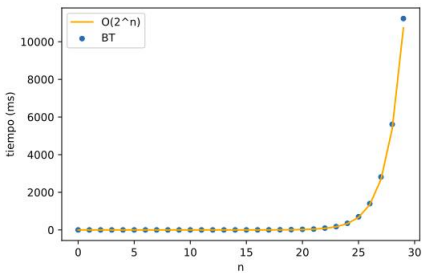
(a) Correlación entre tiempo y complejidad

La curva de los gráficos analizados muestran un comportamiento muy similar a la función a comparar, la cual es 2^n. Esto lo podemos confirmar en la figura 2a con el coeficiente de correlación de Pearson[2] de 0,999975 validando la hipótesis planteada de que la complejidad es efectivamente $\theta(2^n)$. Argumentamos que no hay una correlación del 1.00 por factores externos en las distintas ejecuciones de los experimentos: por ejemplo, el scheduler y/o swapping a disco.

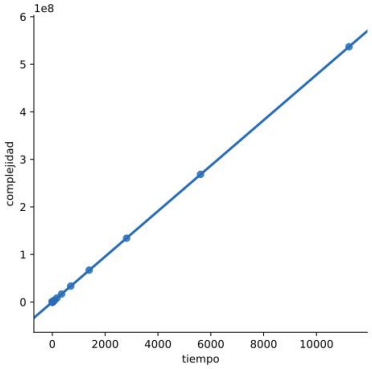
5.3. Experimento 2: Backtracking

5.3.1. Análisis de Complejidad

Nuestra hipótesis es que el método de Backtracking tiene complejidad de peor caso exponencial. Utilizando el dataset de peor caso para la poda de factibilidad, graficamos las instancias de ejecución del algoritmo contra una función exponencial.

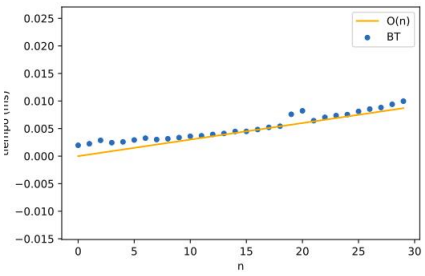


(a) Análisis de complejidad BT-F peor caso

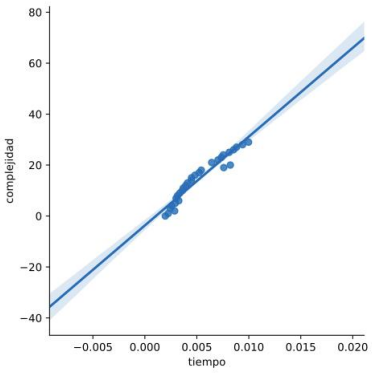


(b) Análisis de coeficiente de Pearson BT-F peor caso

Viendo el gráfico 3a, se puede notar un solapamiento entre BT-F y la función exponencial, demostrando que efectivamente en el peor caso la complejidad es la planteada. Del gráfico 3b, obtenemos que el coeficiente de Pearson es 0.99966, suficiente como para decir que es claramente exponencial la complejidad en peor caso. Otra hipótesis que tenemos, es que la complejidad en mejor caso de la poda por factibilidad es lineal, pues si ningún elemento puede ser metido en el Jambo-tubo sin romperlo, luego sólo recorro la cinta una única vez. Utilizando un dataset de mejor caso para la poda mencionada, la graficamos contra una función lineal.



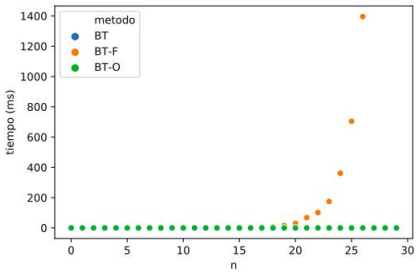
(a) Análisis de complejidad BT-F mejor caso

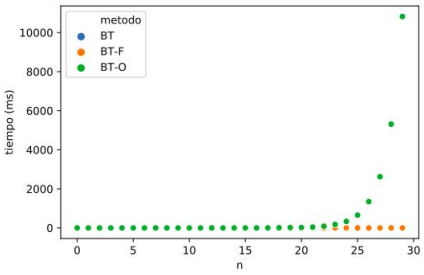


(b) Análisis de correlación BT-F mejor caso

Como podemos apreciar en el gráfico 4a, las instancias graficadas contra la función lineal toman su forma. Lo respaldamos calculando la correlación. La correlación de Pearson en la figura 4b da el valor 0.96517, respaldando nuestra hipótesis de que el mejor caso es de carácter lineal.

5.3.2. Análisis de Podas





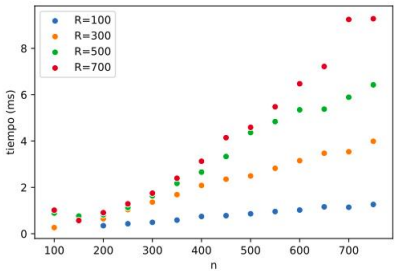
(a) Análisis de podas con densidad alta, k alto. (b) Análisis de podas con densidad baja, k bajo.

En este experimento buscamos comparar la efectividad de la aplicación de las podas de factibilidad y optimalidad del algoritmo de BT. Para ello utilizamos los datasets de densidad alta y densidad baja enunciados en la sección 5.1. La hipótesis es que si tenemos un k óptimo alto (alta densidad) entonces la poda de optimalidad entra en efecto filtrando la mayoría de las ramas, dado que es más probable que exista una rama que no llegue a alcanzar ese k incluso contando todos los elementos restantes. En cambio si tenemos un k óptimo bajo la poda de factibilidad toma mayor importancia al filtrar las ramas que superen el peso o la resistencia individual de alguno de los elementos, dado que es más difícil asegurar que los elementos restantes no superen el k óptimo.

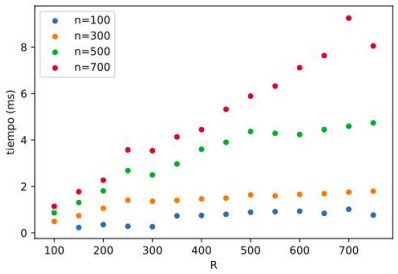
Podemos ver en la figura 5a que en una instancia de densidad alta efectivamente la poda de optimalidad mantiene mejor complejidad temporal que la de factibilidad. También podemos observar en la figura 5b que en una instancia de baja densidad la poda que toma mejor complejidad temporal es la poda de factibilidad, corroborando la hipótesis sobre la efectividad de las podas. Teniendo en cuenta las características de estas podas, no es errado decir que la versión de Backtracking que implementa ambas (en el gráfico, BT, en azul) posee la ventajas de las dos y ninguna de sus inconveniencias.

5.4. Experimento 3: Programación Dinámica

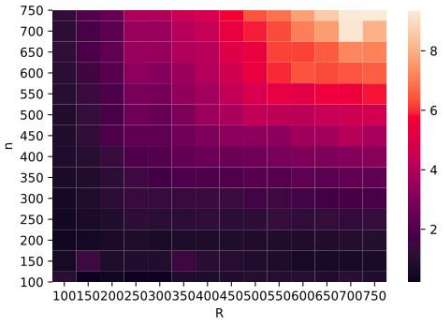
En este experimento sobre la implementación de Programación Dinámica vamos a intentar verificar la cota de complejidad afirmada en la sección 3. Para ello vamos a utilizar el dataset que contiene instancias variando n y R, con resistencias individuales aleatorias.



(a) Tiempo de ejecución en función de n.



(b) Tiempo de ejecución en función de R.



(c) Tiempo de ejecución en función de n y R.

En las figuras 6a y 6b podemos ver como aumenta linealmente el tiempo en función de las variables n y R por separado. En la figura 6c ambas variables entran en juego juntas dando una mejor idea del crecimiento de tiempo lineal en función de n y R en conjunto.

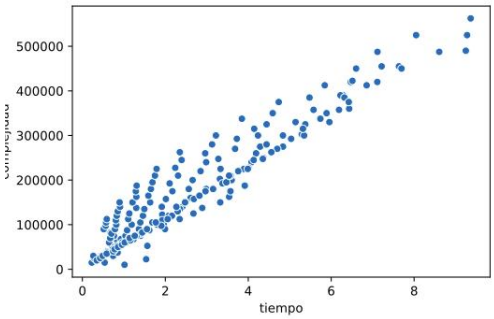
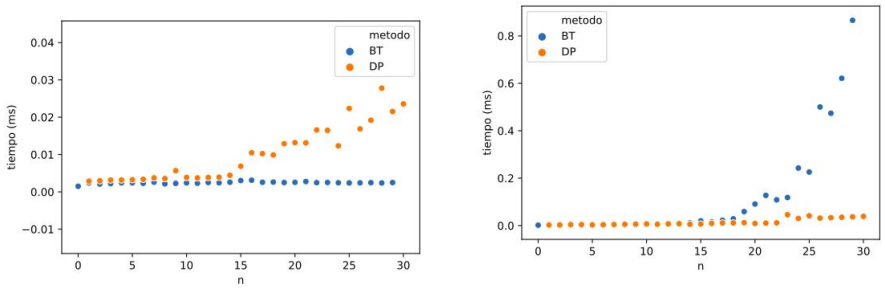


Figura 7: Correlación entre tiempo de ejecución y complejidad.

Finalmente podemos ver la correlación con la complejidad teórica en la figura 7 dando un coeficiente de Pearson[2] de 0.95519. Esto respalda nuestra hipótesis de que efectivamente la complejidad teórica $O(n * R)$ mencionada en la sección 3 es la complejidad del algoritmo de Programación Dinámica implementado.

5.5. Experimento 4: Comparación BT vs PD

Comparamos los tiempos de ejecución de Backtracking completo (usando ambas podas) versus el algoritmo de Programación Dinámica en dos datasets distintos: con densidad alta, y con densidad baja. La hipótesis es que en instancias en que se encuentra un k óptimo alto (se tienen en cuenta más elementos) podemos obtener mejores resultados utilizando BT puesto que se aplicarán las podas, especialmente la de optimalidad. Podemos apreciar en la figura 8a que el algoritmo de



(a) BT vs PD en instancia de densidad alta (b) BT vs PD en instancia de densidad baja

Figura 8: Análisis de PD contra BT

Programación Dinámica no se comporta mejor en el caso en que las instancias posean un k óptimo alto, suponemos que la causa es que la poda de optimalidad filtra la mayoría de las ramas del árbol de recursión. En cambio, en la figura 8b sucede lo contrario: PD tiene mejor rendimiento que BT a medida que aumenta la cantidad de elementos que vienen por la cinta transportadora, puesto que la poda de optimalidad ya no influye como antes.

Razonamos que en instancias en donde se encuentra un k óptimo alto (se tienen en cuenta más elementos) podemos obtener mejores resultados utilizando BT puesto que se aplicarán las podas, mientras que en PD se debe guardar en memoria más estados del programa al considerar muchos elementos. Por otro lado para instancias de densidad baja (donde se tienen en cuenta pocos elementos) vemos el efecto contrario, las podas ya no son tan efectivas en BT, y en PD los estados a guardar en la matriz son menores en cantidad.

6. Conclusiones

Resolvimos el problema de los Jambo-tubos utilizando tres tipos distintos de algoritmos que son Fuerza Bruta, Backtracking, y Programación Dinámica. Experimentamos con diversas instancias representativas que nos permiten verificar las complejidades temporales y características de cada uno, junto con sus ventajas y desventajas.

Pudimos ver que el método de Fuerza Bruta no es viable en ninguna situación, pues tardará tiempo exponencial para cualquier instancia mientras que Backtracking y Programación Dinámica mejoran esa cota de complejidad en casos promedios para resolver el mismo problema. Sin embargo su facilidad de programación nos permitió utilizarlo para verificar los resultados de los métodos más complejos.

El método de Backtracking mejora al método de Fuerza Bruta agregando podas que permiten filtrar ramas, y vimos que estas poseen puntos débiles y fuertes. Pero también vimos que al combinarlas, se tienen todos los beneficios y ninguna de las inconveniencias, dado que se complementan mutuamente.

El método de Programación Dinámica se comporta muy bien con n grande y considerando pocos casos para la solución (k bajo) puesto que evita calcular valores innecesarios, pero pierde contra Backtracking cuando se consideran muchos elementos para la solución (k alto) puesto que debe calcular más valores de la matriz.

Referencias

[1] Enunciado del tp1, . URL https://campus.exactas.uba.ar/mod/resource/view.php?id=103409.

[2] Coeficiente de correlación de pearson, . URL https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.