DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

“Negocio Por Medio”

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Federico Barrena Guzman	236/18	fedebarrena@gmail.com
Camila Camaño	310/17	camicam26@gmail.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

1. Introducción

Distintos sectores del gobierno han decidido evaluar varios métodos de apertura de negocios para evitar el aglomeramiento de gente con el fin de evitar el contagio de coronavirus entre la población. Uno de los proyectos más populares es el llamado *Negocio Por Medio* (NPM).

El proyecto de NPM consiste en maximizar el beneficio de abrir un local de manera tal que el riesgo de expandir la enfermedad no sobrepase cierto límite incontrolable. Más formalmente, dada una secuencia de n locales en orden L = [1,...,n] , con beneficio y contagio $b_i, c_i \in \mathbb{N}$ de cada local $i \in L$, y límite de contagio $M \in \mathbb{N}$. Una solución **factible** de NPM consiste en un subconjunto de locales $L' \subseteq L$ tal que:

(I) $\sum_{i \in L'} c_i \leq$ M ,

(II) No existe $i \in L'$ tal que i+1 $\in L'$.

Consideramos como solución a cualquier subconjunto de L, y será factible si cumple con las condiciones (I) y (II). En este trabajo buscamos además obtener una solución tal que el beneficio sea el mayor posible. Una aclaración es que la solución devuelta será el beneficio máximo encontrado y no el subconjunto de negocios.

A continuación se exhiben algunos ejemplos:
Si $L = \{(10, 10), (30, 20), (15, 20), (50, 40)\}$ con cada elemento representando (beneficio, contagio) de un local, y $M = 50$ entonces la solución óptima tiene beneficio 60 y consiste en abrir los locales $L' =$ { (10,10),(50,40)}. La solución alternativa { (10,10),(30,20),(15,20)} tiene un beneficio mayor pero no es factible ya que no cumple con (II).
Si $L = \{(10, 70), (30, 20), (150, 200)\}$ y $M = 80$ entonces la solución óptima tiene beneficio 30 y consiste en abrir el local $L' =$ { (30,20)}. La solución alternativa { (10,70),(150,200)} tiene un beneficio mayor pero no es factible ya que no cumple con (I).

El objetivo de este trabajo es encontrar soluciones para el proyecto de NPM con tres técnicas de programación distintas y evaluar la efectividad de cada una frente a diferentes instancias. En primer lugar utilzaremos Fuerza Bruta, la cual consiste en enumerar todas las posibles soluciones de manera recursiva y encontrar las factibles entre ellas. Luego, realizaremos un algoritmo de Backtracking realizando podas para reducir el número de nodos visitados en el árbol recursivo. Finalmente, usaremos la técnica de memoización, también conocida como Programación Dinámica, para evitar el solapado de subproblemas.

El orden en el que este trabajo esta detallado es el siguiente:

- Sección 2: algoritmo recursivo de Fuerza Bruta para recorrer todo el conjunto de soluciones y análisis de su complejidad.
- Sección 3: algoritmo de Backtracking con un análisis de mejores y peores casos.
- Sección 4: algoritmo de Programación Dinámica junto con la demostración correspondiente de correctitud y un análisis de complejidad.
- Sección 5: Experimentos computacionales con su respectiva discusión.
- Sección 6: Conclusiones finales.

2. Fuerza Bruta

Un algoritmo de Fuerza Bruta consiste en buscar entre todas las soluciones posibles las que son factibles u óptimas. Para ello, enumera todos los subconjuntos existentes de un conjunto S

y busca las soluciones deseadas. Con respecto a este trabajo, el algoritmo busca la solución en el subconjunto de locales de partes de S (llamado $P(S)$) que permite maximizar el beneficio de locales no adyacentes, sin exceder el límite de contagio M. Por ejemplo:

- si S = { (20,10) , (50, 20) , (30, 40) } y M = 20, el conjunto $P(S)$ = { ∅, { (20, 10) } , { (50, 20) } , { (30, 40) } , { (20, 10) , (50, 20) } , { (20, 10), (30, 40) } , { (50, 20) , (30, 40) } , { (20, 10), (50, 20), (30, 40) } } , y el conjunto de soluciones factibles es { { (50, 20) } , { (20, 10) } } . La solución óptima tiene beneficio acumulado igual a 50 y contagio total de 20.
- si S = { (30, 20) } y M = 10, el conjunto $P(S)$ = { ∅, { (30,20) } } , y el conjunto de soluciones factibles es{ ∅ } . La solución óptima tiene beneficio y contagio acumulado ambos igual a 0.

Lo que se desea hacer en el Algoritmo 1 es recursivamente generar todas las soluciones de S mediante la inclusión (o no inclusión) del beneficio y contagio de cada elemento de S en la solución parcial. El algoritmo guarda el índice del último negocio incluido para saber luego si es que se agrega el beneficio de un negocio adyacente al actual. El bool *esAdyacente* permite saber si en algún momento se agregó un negocio contiguo.

Una vez que se llega a una hoja, el algoritmo determina si la solución encontrada es factible, es decir, chequea si el contagio acumulado es menor o igual al límite indicado y si nunca hubo una inclusión de negocios contiguos. De ser factible, devuelve el beneficio acumulado.

Indicamos con -∞ que no hay solución. Representamos la secuencia de negocios con S, el límite de contagio con M, i y j indices de S, b y c el beneficio y contagio de cada local respectivamente, y esAdyacente es un bool.

Algorithm 1 Algoritmo de Fuerza Bruta para SSP.

```
1: function FB(S, M, i, j, b, c, esAdyacente)
2:     if i = n then
3:         if c ≤ M and ¬ esAdyacente then
4:             return b
5:         else
6:             return -∞
7:     bool nuevo ← false
8:     nuevo ← i - j = 1
9:     return máx{FB(i + 1, i, b + S_i.first, c+S_i.second, esAdyacente or nuevo),
10: FB( i+1, j, b, c, esAdyacente) }.
```

La correctitud del algoritmo se basa en que se generan todas las posibles soluciones. En cada elemento de S se crean dos ramas donde en una se considera agregar un negocio al beneficio y su contagio asociado y en otra no. Por ello, la óptima solución es encontrada si existe.

La complejidad del Algoritmo 1 para el peor caso es $O(2^n)$ debido a que el árbol de recursión es un árbol binario completo de $n + 1$ niveles. En cada nodo se ramifica en dos hijos y en cada paso el parámetro i es aumentado en 1 hasta llegar a n. El parámetro j siempre será menor o igual a i y no es utilizado para recorrer, por lo que no impacta en el avance del algoritmo.

Al tratarse de un vector de tuplas, el acceso a cada elemento, ya sea de la tupla o del vector, es de tiempo constante. Con lo cual, es importante observar que la solución de cada llamado recursivo toma tiempo constante al tener las lineas 2 . . . 11 con operaciones elementales. Como corolario, se puede concluir que el algoritmo se comporta de igual manera frente a todos los tipos de instancia, dado que siempre genera el mismo número de nodos, como se observa más adelante en la sección de experimentos. Coloquialmente decimos que el conjunto de instancias de peor caso es igual al conjunto de instancias de mejor caso.

3. Backtracking

Los algoritmos de Backtracking consisten en enumerar todas las soluciones formando un árbol de backtracking de manera similar a Fuerza Bruta. La diferencia reside en que hay **podas** en el árbol para evitar recorrer ramas donde ya se sabe que no habrá una solución que cumpla con lo pedido. Las podas más comunes se dividen en dos categorías: factibilidad y optimalidad.

1. **Poda por factibilidad:** Consiste en recorrer el árbol teniendo en cuenta las instancias que cumplen con las condiciones del problema y descartar las que no. Más específicamente, consiste en dejar de extender aquellas soluciones parciales donde el contagio acumulado $c >$ M. De esta manera, reducimos la cantidad de operaciones que realiza el algoritmo. Está en la línea 12 del algoritmo 2.

2. **Poda por optimalidad:** Similarmente a la poda por factibilidad, la idea de esta poda consiste en encontrar aquella (o aquellas) solución/es que cumplan lo pedido y que además sean las mejores posibles. Para este problema, sea V el mayor beneficio encontrado hasta el momento y sumRes la suma restante ya precalculada. Vamos a suponer que estamos en un nodo intermedio que representa una solución parcial con beneficio b. En este caso, si b + sumaRestante $\leq$ V, nos podemos asegurar que haya de haber una solución mayor a V, no puede estar en esa rama del árbol y por ende nos podemos ahorrar cómputo de operaciones al no seguir explorando por esa rama. Éste se encuentra en la línea 13 del algoritmo 2.

Ambas podas se pueden implementar en este caso ya que cumplen con el efecto *dominó*, es decir; si se llega a una solución que superó el límite de contagio, nos podemos asegurar que cualquier otra solución en esa rama no va a ser posible dado que sobrepasa el contagio acumulado. Con respecto al beneficio, tenemos la máxima suma hasta en un momento y una *suma restante* ya precalculada en $O(n)$ que sirve para poder comprobar si una rama contiene un potencial máximo beneficio o no, y podar de acorde.

Algorithm 2 Algoritmo de Backtracking para NPM.

```
1:  V ← −∞
2:  sumRes ← maxSum(S)
3:  function BT( S, M, i, j, b, c, esAdyacente, sumRes)
4:      if i = n then
5:          if c ≤ M ∧ ¬esAdyacente ∧ b > V then
6:              V ← b
7:          if (c ≤ M ∧ ¬ esAdyacente) then
8:              return b
9:          else
10:             return −∞
11:     bool nuevo ← i - j = 1
12:     if c > M then return -∞                          ▷ Poda por factibilidad
13:     if b + sumRes ≤ V then return - ∞                ▷ Poda por optimalidad
14:     return máx { BT( i+1, i, b + S_i.first, c+ S_i.second, esAdyacente ∨ nuevo, sumRes -
        S_i.first), BT( i+1,j, b,c,esAdyacente, sumRes - S_i.first) }.
```

La complejidad del algoritmo en peor caso es $O(n + 2^n) = O(n) + O(2^n)$, donde fácilmente se puede deducir que termina perteneciendo a $O(2^n)$. El peor escenario es en el que no se realiza ninguna poda, por lo que debe recorrer todas las posibles soluciones y termina siendo una situación igual a Fuerza Bruta. Una observación importante es que en las líneas 4 ... 13 se están haciendo operaciones constantes de comparación, suma y asignación, con lo cual no deben impactar en el tiempo de ejecución. Las instancias en donde las podas terminan siendo no efectivas (es decir, se recorre todo el arbol) son del tipo S = (1, 1) ... (1, 1), en donde M = n = |S| . Esto es debido

a que al no poder agregar negocios adyacentes, el contagio acumulado nunca llega a superar a M por lo que siempre recorrerá todos los locales hasta llegar al final del árbol.

Hay instancias donde el algoritmo se comporta de manera lineal, especfícamente cuando $S = \{(b, c), \ldots, (b, c), (b, m)\}$ donde $b > 0$, $c > M$ y $m < M$. En este caso, el algoritmo explora la primera rama y poda por factibilidad todas aquellas que superan el valor de M, garantizando que ninguna rama sea explorada. Esto efectivamente sería una de las familias de instancias que pertenecen al mejor caso de Backtracking, con una complejidad de $O(n)$. Esto se debe a que recorre cada elemento del vector una vez, descartándolos a todos menos el último elemento, con lo que termina evaluando todos los elementos de S una única vez.

4. Programación Dinámica

Programación Dinámica (PD) no sólo sirve en problemas de optimización, aunque es cierto que la mayoría de los problemas en los que aparece son de este tipo. En el contexto de un problema de optimización, para que exista una estructura recursiva es necesario que se cumpla el Principio de Optimalidad. Esto es, el problema debe poder resolverse utilizando unicamente soluciones optimas a subproblemas. En el contexto de este problema es útil ya que nos permite saltear el cómputo de un subárbol si éste ya fue previamente calculado en otro lado. Nuestro problema puede definirse como la función:

$$f(i, j, c, adyacencia) = \begin{cases} -\infty & si\ c > M, \\ -\infty & si\ adyacencia, \\ 0 & si\ i = n \land \text{c} \leq \text{M}, \\ max\{\ f(i+1, j, c, adyacencia), & \\ S_i\text{.first + f(i+1, i, c + S}_i\text{.second, adyacencia)}\} & \text{en caso contrario.} \end{cases}$$

Donde nuestro f(i, j, c, adyacencia) se define coloquialmente como "máxima suma de beneficios de S cuyo contagio no supera M y no hay negocios adyacentes".

Correctitud

(I) Si $c > M$ entonces claramente ningún local va a sumar a $M - c < 0$ ya que el contagio de cada local es entero positivo, así que la única respuesta posible es $f(i, j, c, adyacencia) = -\infty$.

(II) Si $adyacencia = True$ entonces quiere decir que se incluyeron 2 o más locales contiguos, lo que contradice las condiciones originales de nuestro problema. Por lo tanto, la respuesta es $f(i, j, c, adyacencia) = -\infty$.

(III) Si $i = n$ y $c \leq M$ entonces quiere decir que ya evaluamos todos los locales y encontramos una solución factible, ya que si se hubiese superado el contagio entonces caería en la primera definición. Como ya se llegó al final de S, no hay más locales disponibles y la respuesta tiene que ser 0.

(IV) En este caso, $i \leq n$ y $c \leq M$ entonces estamos efectivamente buscando un subconjunto de locales de $S^i = \{S_i, \ldots, S_n\}$ que sume $M' = M - c \geq 0$. De existir alguno, tiene que o bien tener al i-ésimo local (o elemento) o no tenerlo. Si no lo tiene, entonces tiene que ser a su vez un subconjunto de S^{i+1} y sumar M', por lo tanto, debe encontrarse de manera recursiva $f(i + 1, j, c, adyacencia)$. Si tiene al i-ésimo elemento, entonces el resto de la solución debe sumar a lo sumo $M' - S_i$, utilizando elementos de S^{i+1} y debe ser la de máximo beneficio entre todas ellas. Esto es precisamente $f(i + 1, j, c + S_i, adyacencia)$. Por lo tanto, la mejor solución es f(i, j, c, adyacencia) = máx$\{f(i + 1, j, c, adyacencia), S_i$.first + f(i+1, i, c + S_i.second, adyacencia) $\}$. Notar que al término de la derecha se le suma el beneficio de S_i por haber seleccionado al i-ésimo elemento.

4.1. Memoización

Se puede observar que al calcular el valor intermedio en un nodo, éste podría necesitar el uso del anterior, y esta instancia a su vez posiblemente use el nodo precedente. Ahora bien, tambien podría pasar que un nodo en otra rama tenga que disponer de un nodo ya previamente calculado en otro lado. Esto muestra que es un subproblema compartido, y es evidente que lo mismo sucede para subproblemas más pequeños. Tenemos entonces una formulación recursiva y los subproblemas se solapan. Entonces PD es una buena idea para resolver este problema.

Si usamos una memoria tal que recuerde el resultado de un caso ya resuelto ya no será necesario recalcularlo nuevamente. A esta memoria se la puede pensar como una matriz de $N + 1$ x $M + 1$ elementos. Así, nos aseguramos de no resolver más que nM casos. En la línea 12 del Algoritmo 3 se ve esta idea, donde se ejecuta sólo si el estado previo no habia sido calculado.

Algorithm 3 Algoritmo de Programación Dinámica para SSP

```
1: mem_iw ← ⊥ for i ∈ [1, n], m ∈ [0, M].
2: function DP(i, j, b, c, esAdyacente)
3:     if c > M then
4:         return -∞
5:     if adyacencia then
6:         return -∞
7:     if i = n and c <= M then
8:         return 0
9:     bool hayAdyacencia ← i - j = 1
10:    if mem[i][c] = ⊥ then
11:        mem[i][c] = max{PD(i+1, j, c, adyacencia),
12: S_i.first + PD(i+1, i, c + S_i.second, adyacencia or hayAdyacente)}
13:
```

5. Experimentación

En esta sección vamos a presentar los experimentos computacionales sobre los distintos métodos presentados en las secciones anteriores. Las ejecuciones fueron realizadas en una workstation con CPU Intel Celeron @ 1.5 GHz y 4 GB de memoria RAM, y utilizando el lenguaje de programación $C++$.

5.1. Métodos

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

- **F**B: Algoritmo 1 de Fuerza Bruta de la Sección 2.
- **B**T-F: Algoritmo 2 de Backtracking de la Sección 3, considerando únicamente podas por factibilidad.
- **B**T-O: Algoritmo 3 de Backtracking de la Sección 3, considerando únicamente podas por optimización.
- **D**P: Algoritmo 4 de Programación Dinámica de la Sección 4.

5.2. Instancias

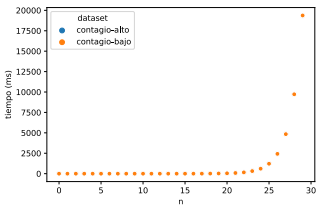
Para evaluar los algoritmos en distintos escenarios definimos familias de instancias de características diferentes entre sí. Nuestro objetivo es observar qué pasa para valores variados de contagio y beneficio en los negocios en comparación con M. El n de cada instancia representa la cantidad de negocios. Los datasets definidos se enumeran a continuación.

- **contagio-alto**: Esta familia de instancias está formado por negocios de muy alto contagio comparado con el valor de M. Los datos están compuestos por números $1, \ldots, n$ en S en orden aleatorio para el contagio y beneficio, y tomamos $M = \frac{n}{2}$. La idea es que con poca cantidad de negocios se pueda llegar a un valor muy cercano o igual a M.
- **contagio-bajo**: Para esta conjunto de instancias se toman los números $1, \ldots, n$ en S en algún orden aleatorio para el contagio y beneficio de cada negocio, y se toma M = n*4 . El objetivo es que se puedan admitir muchos negocios de bajo contagio.
- **bt-mejor-caso**: Cada instancia de n negocios está formada por $S = \{(1001, 1001), \ldots, (1001, 1)\}$ y M = 1000. Son las instancias para el mejor caso de Backtracking definidas en la Sección 3.
- **bt-peor-caso**: Cada instancia de n negocios, está formada por $S = \{(1, 1) \ldots, (1, 1), (1, 1)\}$ y $M = 1000$. Son las instancias para el peor caso de Backtracking definidas en la Sección 3.
- **sin-contagio**: Estas instancias están compuestas por n negocios, con $S = \{(b, 1), \ldots, (b, 1)\}$, donde b va de 1 ... n y el contagio es siempre igual en todos los negocios. La idea es ver qué pasa con la poda de optimalidad de BT, la cual poda según beneficio y contagio.
- **beneficio-igual-contagio-alto**: Estas instancias están compuestas por n negocios, donde $S = \{(b, c_1), \ldots, (b, c_n)\}$, donde b es igual para todos los negocios. El rango de c es 1 ... n.
- **dinamica**: Esta familia de instancias tiene datos con combinaciones de valores para n, los contagios, beneficios y M en los intervalos $[1000, 8000]$. Los números en S son una permutación de el conjunto $\{1, \ldots, n\}$.
- **m-alto**: En esta famila de instancias, el valor de M es mucho mayor en comparación con el valor de n, formalmente $M = n * 4$. Lo utilizaremos para comparar el comportamiento de Programación Dinámica frente a Backtracking. Los valores del beneficio y contagio de cada negocio están en el rango $1, \ldots, n$.
- **m-bajo**: En esta famila de instancias, el valor de M es mucho menor en comparación con el valor de n, más formalmente $M = \frac{n}{4}$. Los valores del beneficio y contagio de cada negocio están en el rango $1, \ldots, n$.

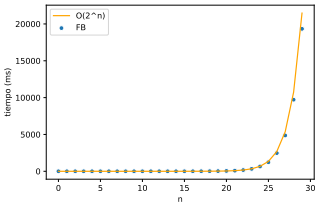
5.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento vamos a analizar cómo se comporta el método FB en distintos contextos. El análisis de complejidad de la Sección 2 afirma que la complejidad temporal para el mejor y peor caso es igual y exponencial en función de n para ambos. Para verificar empíricamente estas ideas vamos a evaluar FB utilizando los datasets contagio-alto y contagio-bajo, y graficar los tiempos de ejecución en función de n.

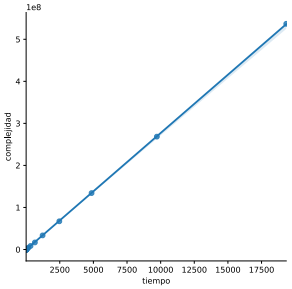
La Figura 1a presenta los resultados del experimento, en donde las curvas para las instancias de contagio alto y contagio bajo están solapadas. Lo que nos dice este gráfico es que los tiempos de ejecución no cambian según la instancia.



(a) Tiempo de ejecución del método FB sobre contagio-alto y contagio-bajo.



(b) Tiempo de ejecución contra complejidad esperada.



(c) Correlación entre el tiempo de ejecución y la complejidad esperada.

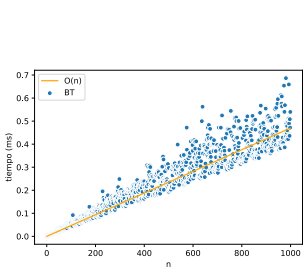
Figura 1: Análisis de complejidad del método FB.

Vamos a ver la correlación de la complejidad estudiada en la Sección 2, $O(2^n)$, y entre los datasets. En la Figura 1b se muestra el tiempo de ejecución de FB a la par de una función exponencial de $O(2^n)$. En la Figura 1c se enumeran las instancias $I_1, \ldots, I_k$ y para cada una se grafica el tiempo de ejecución real $T(I_i)$ contra el tiempo esperado $E(I_i) = 2_i$, es decir, su gráfico de correlación.

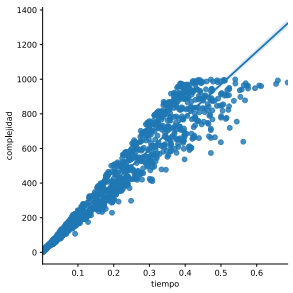
Podemos ver que el tiempo de ejecución acompaña la curva exponencial y que la correlación con la función 2^n también lo hace. Además, el índice de correlación de Pearson de ambas variables es $r \approx 0{,}999992$. Por ende, podemos afirmar que el algoritmo tiene un comportamiento como el inicialmente propuesto en las hipótesis.

5.4. Experimento 2: Complejidad de Backtracking

En esta experimentación se contrasta las hipótesis de la Sección 3 con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método BT con respecto los datasets bt-mejor-caso y bt-peor-caso.



(a) Tiempo de ejecución vs Complejidad esperada.

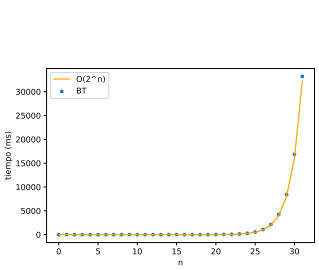


(b) Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada.

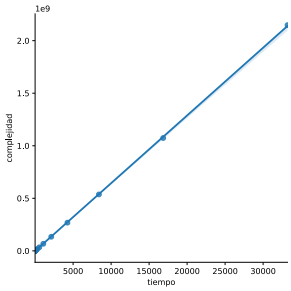
Figura 2: Análisis de complejidad del método BT para el data set bt-mejor-caso.

Las Figuras 2 y 3 muestran los gráficos de tiempo de ejecución de BT y de correlación para cada dataset respectivamente. Como habíamos conjeturado, las hipótesis presentadas anteriormente se cumplen para ambos casos. Por un lado, para las instancias de mejor caso se puede ver que nuestro algoritmo con distintas entradas muestra un crecimiento lineal, aunque no perfecto. Una posible razón para esta anomalía es que al ser un comportamiento lineal, los tiempos de ejecución son muy bajos, incluso para nuestros valores de entrada. Como resultado, cualquier interferencia indeseada de nuestro sistema operativo puede alterar los resultados. Sin embargo, el índice de correlación de Pearson es $r \approx 0{,}94536$ lo cuál muestra que hay una correlación positiva fuerte entre los tiempos de ejecución y una función lineal.

Por otra parte, para las instancias de peor caso no se ve este comportamiento, y los tiempos de ejecución se presentan más ajustados a la curva de complejidad exponencial, ya que el tiempo de ejecución más grande no es tan afectado por cambios de contexto. Notemos que en este caso se ejecutaron instancias hasta $n = 30$, número elegido para evitar que el tiempo de ejecución sea demasiado grande (> 10 minutos). Para estas instancias el índice de correlación de Pearson es de $r \approx 0{,}99999927$ contra una función exponencial con base 2, por lo que se puede asumir fuertemente que efectivamente tiene un comportamiento exponencial.



(a) Tiempo de ejecución vs Complejidad esperada.



(b) Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 3: Análisis de complejidad del método BT para el data set bt-peor-caso.

5.5. Experimento 3: Efectividad de las podas

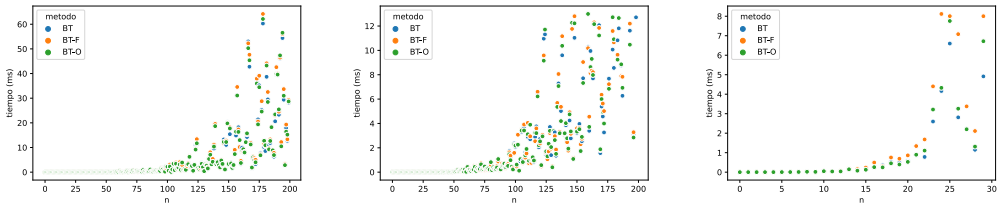
Como ya vimos en la experimentación anterior, no cabe duda que las podas son esenciales cuando se trata de reducir el tiempo de búsqueda. La incógnita que tenemos hasta el momento es qué sucede en el medio del comportamiento lineal y exponencial, y qué factores afectan al algoritmo para ir transitando entre ambos escenarios. Una de las hipótesis es que el Algoritmo 2 mejora su funcionamiento dependiendo del contagio de las instancias, es decir, de cuántos elementos de la segunda componente de la tupla de los elementos de S se necesitan para no exceder M. Por ejemplo, si $M = 100$ y $S_i[1] > \{ b, 50\}$ para todo i, para todo b ≥ 1, entonces al seleccionar dos elementos en una solución parcial del algoritmo, o bien se suma M o bien se supera ese valor y la poda por factibilidad es ejecutada, sin importar el b. Por otra parte, esto también incide en el funcionamiento de la poda por optimalidad ya que al encontrarse una solución cuyo contagio supera el contagio restante disponible, si además el beneficio es menor al beneficio más grande encontrado, se efectúa la poda por optimalidad.

En este experimento se compara el funcionamiento de los métodos BT, BT-F y BT-O con respecto a los datasets contagio-alto, contagio-bajo, beneficio-igual-contagio-alto y sin-contagio. La hipótesis es que para las instancias de alto contagio los algoritmos van a ser más eficientes que con aquellas de bajo contagio.

En la Figura 4a se muestra los resultados para el dataset contagio-alto. Se ejecutaron instancias hasta $n = 200$ para evitar que el algoritmo demore más de unos minutos. Dicho esto, se puede observar que los tiempos de ejecución para los tres métodos están entre aquellos observados para los mejores y peores casos del algoritmo de BT. Por otra parte, se puede observar que para todos los casos las podas por factibilidad no parecen efectuarse tan seguido, ya que las últimas instancias de tamaño más grande apuntan a que BT-F crece más rapidamente que los otros 2 algoritmos. Esto puede deberse al hecho de que el contagio fue definido con respecto al máximo contagio de un elemento en S pero sin tener en cuenta el resto, por lo tanto, puede ser que existan varios elementos que juntos sean menores a M afectando la efectividad de la poda por factibilidad. La Figura 4b hace un acercamiento para evaluar mejor la diferencia entre BT y BT-O. En ella se aprecia que la combinación de ambas podas es ligeramente más efectiva y es por esto que se puede concluir que la poda por factibilidad tiene un impacto positivo en el algoritmo final. Sin embargo, la diferencia en términos de complejidad es casi insignificante como para que haya una distinción clara entre ambas podas.

Por el lado de las instancias de bajo contagio, los resultados están expuestos en la Figura 4c. El mensaje principal de estos gráficos es que similarmente a las instancias de contagio alto, la efectividad de las podas por factibilidad no ocurre lo suficientemente seguido como para modificar de manera considerable su comportamiento que parecería ser exponencial. En particular, esto se puede observar a partir de $n = 15$ que si bien presenta tiempos más chicos que en las instancias de peor caso conserva su naturaleza exponencial. Cuando se trata de poda por optimalidad, su

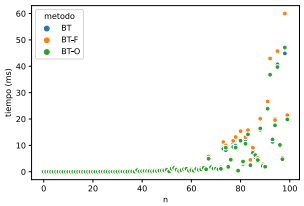
comportamiento no cambia, lo que podría indicar que el beneficio incide más en la poda. Esto puede deberse a que como cada local tiene un contagio que es menor a M y BT-O evalúa sólamente el beneficio, para que mantenga su comportamiento lineal, BT-O tiene que estar podando en base al beneficio. Como conclusión podemos afirmar que el contagio de las instancias no tiene un peso significativo, o mas bien decisivo, en el funcionamiento de este algoritmo.



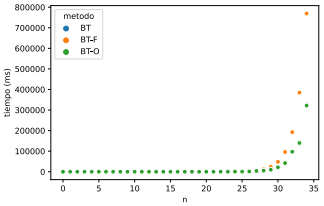
(a) Efectividad de las podas para contagio-alto.

(b) Efectividad de las podas con zoom para contagio-alto.

(c) Efectividad de las podas para contagio-bajo.



(d) Efectividad de las podas para negocios de beneficio igual y contagio variado.



(e) Efectividad de las podas para negocios de beneficio variado y contagio igual.

Figura 4: Comparación de efectividad en las podas.

Para los casos donde el beneficio es el mismo en todos los negocios mostrados en la Figura 4d vemos que ambas podas tienen un comportamiento exponencial similar. En este caso, optimalidad no realiza ninguna poda según el beneficio de los negocios por lo que actúa de manera similar a factibilidad. Es por ello que razonamos que el beneficio no tiene gran impacto en la poda de optimalidad.

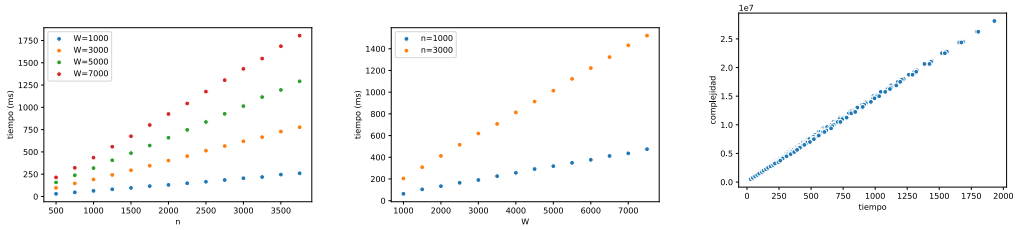
En la Figura 4e podemos ver que la poda por optimalidad es ligeramente mejor que la poda por factibilidad. Sin embargo, esta diferencia es muy pequeña como para afirmar con seguridad que verdaderamente optimalidad es mejor que factibilidad. Otra observación es que este es el caso donde las podas se comportan de manera más parecida. Por ende, concluimos que un contagio igual en todos los negocios hace que las podas actuen con la misma naturaleza ya que nunca se realiza ninguna poda por contagio. La única poda realizada es por beneficio en optimalidad, pero el cambio en su comportamiento en comparación con factibilidad es despreciable.

En resumen, luego de analizar diferentes casos donde se varia el beneficio y contagio de los negocios, concluimos que no influyen fuertemente en la complejidad temporal de las podas, y para que haya una distinción notable, el tamaño de la instancia tiene que ser muy grande (¿200).

5.6. Experimento 4: Complejidad de Programación Dinámica

A continuación vamos a analizar la eficiencia del algoritmo de Programación Dinámica y su correlación con la complejidad teórica calculada en la Sección 4. Para ello, se ejecutan las instancias del dataset dinámica sobre DP y se grafican los resultados en la Figura 5.

Las Figuras 5a y 5b muestran el avance del tiempo de ejecución en función de n y M respectivamente. Ambas figuras muestran que los puntos tienen un crecimiento lineal en función de ambas variables.Por último, el gráfico de correlación reafirma que hay una correlación positiva entre ambas series de datos. El índice de correlación de Pearson es de $r \approx 0{,}99951$.



(a) Tiempo de ejecución en función de n.

(b) Tiempo de ejecución en función de M.

(c) Correlación entre el tiempo de ejecución y la cota de complejidad temporal.

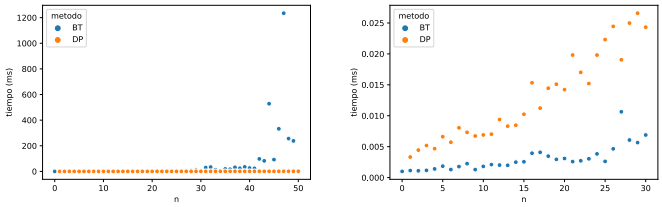
Figura 5: Resultados computacionales para el método DP sobre el dataset dinámica.

5.7. Experimento 5: Comparación Backtracking y Programación Dinámica.

Como último experimento, vamos a compararar dos técnicas algorítmicas distintas. El objetivo es determinar para qué situaciones es útil utilizar cierto algortimo ante otro, y además ver su comportamiento lado a lado. Nosotros pensamos que en instancias donde el contagio M es muy alto y la cantidad de negocios es baja en comparación, BT va a funcionar mejor gracias a sus podas, además de que será más barato de mantener que una estructura de memoización. Mientras tanto, DP será ideal para situaciones donde el valor de M es bajo comparado con la cantidad de negocios.

En el primer gráfico 6a podemos ver que BT mantiene su comportamiento exponencial y DP lineal cuando M es considerablemente mayor que n. Mientras tanto, el segundo gráfico 6b muestra un comportamiento inverso: BT se comporta linealmente mientras que DP va aumetando en su complejidad cuando M es mucho menor que n. Estas observaciones logran confirmar la hipótesis enunciada anteriormente.

Algo a tener en cuenta al usar DP es que el uso de memoria de este algoritmo ampliamente supera al de Backtracking, ya que para cada instancia se tiene que computar una matriz de ($N+1$ x $M+1$) elementos. Por lo tanto, el uso de cada algoritmo va a estar sujeto al contexto en el que se lo quiera implementar. Por ejemplo, si se tuviera un límite de memoria acotado, BT será el algoritmo más conveniente. Ahora si lo que se busca es eficiencia en términos de complejidad temporal, DP será la mejor opción, ya que no se computan las mismas ramas más de una vez.



(a) Dataset donde M >>> n.

(b) Dataset donde M <<< n.

Figura 6: Comparación de tiempos de ejecución entre DP y BT.

6. Conclusiones

En este trabajo presentamos tres algoritmos para resolver la problemática propuesta por el gobierno. El algoritmo de Fuerza Bruta siempre se comporta con la misma complejidad en cualquier caso por lo que es poco eficiente e incontrolable. Las podas de Backtracking logran reducir la cantidad de soluciones exploradas, y entre ellas no hay una diferencia considerable. Además, sus

podas no se ven afectadas por el valor del contagio y beneficio de los negocios. Mientras tanto, si el valor límite M es muy grande en comparación con n, Programación Dinámica logra reducir la complejidad del problema a lineal. Otro trabajo a futuro podría ser correr experimentos en base a la complejidad espacial de los algoritmos, poniendo un énfasis en PD, ya que los experimentos realizados en este Trabajo Práctico no tuvieron en cuenta la cantidad de memoria que requiere cada algoritmo y es un atributo que es de mucho interés.