



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
x x	x/x	x@x.x
Lucas Fabrizio Di Salvo	446/18	lucasdisalvo@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

Debido a la situación actual de pandemia, para evitar la rápida propagación del virus, se utiliza el reconocido distanciamiento social que indica que dos personas no deben estar a menos de dos metros de distancia entre sí, por lo cual, gran parte de los locales y negocios de las principales avenidas de la ciudad no pueden reabrir sus puertas. Es por esto que se creó el proyecto Negocio Por Medio (NPM), que intenta dar una alternativa para abrir los locales que mayor beneficio generan para la economía, de tal forma que se controle la propagación de la enfermedad.

Consiste en asignar a cada local de las grandes avenidas dos valores, uno que representa el beneficio que genera para la economía, y otro que indica el valor de contagio que puede producir con su apertura. La idea es determinar, dada una avenida de la ciudad, cuáles locales abrir tal que no queden dos locales consecutivos abiertos, que no se exceda el límite de contagio en los locales abiertos y que se maximice el beneficio total.

Formalmente, sea una secuencia de n locales $L = [1, \dots, n]$, el beneficio y contagio del local $i \in L$, $b_i, c_i \in \mathbb{N}_{\geq 0}$ y el límite de contagio $M \in \mathbb{N}_{\geq 0}$, una solución factible consta de un conjunto de locales $L' \subseteq L$ tal que:

- $\sum_{i \in L'} c_i \leq M$
- $\nexists i \in L'$ tal que $i + 1 \in L'$

Para el desarrollo a continuación, se tienen en cuenta los siguientes ejemplos

- $L = [1, 2, 3, 4]$ con $B = [50, 25, 10, 20]$, $C = [10, 20, 20, 30]$ y un límite de contagio $M = 40$ donde la solución factible óptima es 70.
- $L = [1, 2, 3, 4, 5]$ con $B = [20, 20, 30, 10, 40]$, $C = [10, 30, 10, 20, 20]$ y un límite de contagio $M = 50$, siendo 90 la solución factible óptima.

El objetivo de este trabajo es abordar el problema de NPM a través de tres técnicas de programación: Fuerza Bruta, Backtracking y Programación Dinámica; y evaluar la efectividad de cada una en la resolución de este problema para distintos conjuntos de instancias.

El informe esta dividido en secciones: La Sección 2 esta dividida a su vez en 3 sub secciones, una para cada técnica utilizada (Fuerza Bruta, Backtracking y Programación Dinámica) donde en cada una se encuentran las definiciones de los algoritmos realizados, junto con ejemplos de instancias y soluciones válidas, además del análisis de complejidad de los mismos. Luego en la Sección 3 se encuentran los experimentos computacionales realizados y su respectivo análisis. En la Sección 4 damos la conclusión final, y por último en la Sección 5 se encuentra el apéndice.

2. Técnicas Algorítmicas

2.1. Fuerza Bruta

Un algoritmo de **Fuerza Bruta** consiste en enumerar todas las posibles soluciones para un problema dado, buscando aquellas que sean *factibles* u *óptimas* dependiendo del problema tratado.

En este caso, el conjunto de soluciones está compuesto por todos los subconjuntos de L (es decir, su conjunto de partes con 2^n elementos, notado $\mathcal{P}(L)$), de los cuales puede que algunos de ellos sean soluciones *factibles* para nuestro problema. En caso de haberlas nos interesa la que consideremos *óptima*, en otras palabras, la que provea un mayor beneficio.

La idea del Algoritmo 1 para resolver el problema es ir generando las soluciones de manera recursiva decidiendo en cada paso si un local de L está abierto o no y quedándose con la mejor solución de alguna de las dos ramas (la que provea mayor *beneficio*). Finalmente, al identificar una solución, determinar si es *factible* (no excede el límite de *contagio* y sus locales están distanciados), de ser así, devolver su beneficio.

Algorithm 1 Algoritmo de Fuerza Bruta

```

1: function  $FB(n, B, C, M, i, b_{aux}, c_{aux}, indices)$ 
2:   if  $i == n$  then
3:     if  $c_{aux} \leq M \wedge distanciados(indices)$  then
4:       return  $b_{aux}$ 
5:     else
6:       return  $-\infty$ 
7:   return  $\max\{FB(n, B, C, M, i + 1, b_{aux}, c_{aux}, indices),$ 
8:            $FB(n, B, C, M, i + 1, b_{aux} + B_i, c_{aux} + C_i, indices + +[i])\}$ 

```

Para el primer ejemplo, se puede ver el árbol de recursión sobre $FB(4, b, c, M, 0, 0, \{\})$, donde en cada nodo se distingue una tupla con el beneficio como primer valor y el nivel de contagio como segundo. Para el conjunto de datos dado, la solución óptima es $\{1, 4\}$, cuyo beneficio es de 70, respetando que el contagio no exceda el M dado (40 en este caso). En la ejecución original para los valores dados, se inicia con 0 como beneficio y 0 como contagio, puesto que no hay ningún local considerado para la solución aún, de esta manera, el algoritmo no efectuará su caso base, sino que realizará el caso recursivo, donde por un lado descarta el primer local, y por el otro, lo considera para la solución, se puede apreciar el que la solución óptima sí lo considera. De esta manera sucederán ejecuciones recursivas donde se generán todas las soluciones posibles.

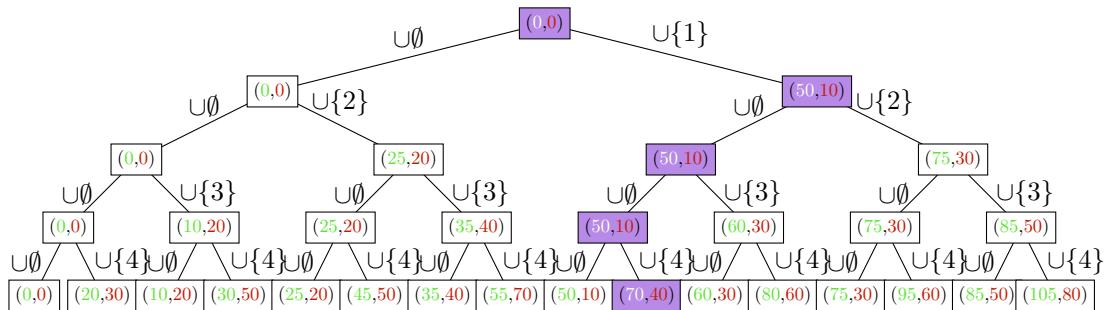


Figura 1: Ejemplo de ejecución del Algoritmo para $n = 4$, $b = [50, 25, 10, 20]$, $c = [10, 20, 20, 30]$ y $M = 40$. En violeta la solución óptima $L' = \{1, 4\}$.

La razón por la cual $L' = \{1, 4\}$ es la solución óptima es la siguiente, se quiere que no haya dos locales contiguos, es por esto que no es una solución *factible* el tomar todos los locales ($L'' = \{1, 2, 3, 4\}$), por más que produzca el mayor beneficio. Por otro lado, no se quiere exceder el nivel de contagio, con lo que por esta razón L'' tampoco sería una solución factible, y adicionalmente, como se quiere el mayor beneficio respetando estas condiciones, L' es la que provee el mayor de ellos. De chequear esta condición, se ocupa la función *distanciados*¹.

En el segundo ejemplo, donde se tiene $n = 5$, la solución óptima sería $L = \{1, 3, 5\}$ obteniendo un beneficio de 90 con un contagio igual a 40.

La complejidad del algoritmo en peor caso es $O(n \times 2^n)$ dado que el árbol de recursión es de tipo binario y tiene $n + 1$ niveles (n niveles + la raíz) ya que cada nodo posee 2 hijos que representan el llamado recursivo en el cual se abre el local o no y se llama a la función incrementando el i hasta llegar a n . Cada paso recursivo tiene costo lineal debido a concatenar el i a *índices*, y pasarlo por copia a la función. Además cuando se llega al caso base, en otras palabras, no hay mas locales posibles por recorrer, el algoritmo revisa si los locales que tomó están distanciados entre sí, esto se hace a través de la función *distanciados*, la cual tiene costo $O(n)$.

Puede notarse que el algoritmo reacciona siempre de igual manera sin importar la entrada, es decir, su árbol siempre generará $2^{n+1} - 1$ nodos.

2.2. Backtracking

Los algoritmos de **Backtracking** son similares a los de Fuerza Bruta salvo que, en vez de enumerar todas las soluciones posibles (mas allá de que sean válidas o no), se evita explorar partes del árbol de recursión en las que ya se sabe que no va a poder existir alguna solución que forme parte del conjunto de soluciones válidas u óptimas. Esto se hace mediante las *podas*. Las más comunes y las que utilizaremos a continuación son las de *factibilidad* y *optimalidad*.

Poda por factibilidad Como queremos obtener el máximo beneficio posible producido por un subconjunto de locales de L que a su vez la suma de los riesgos de cada uno no supere el *límite de contagio*, sea L^i una

¹Debido a que se consideró que no era relevante al análisis de FB, su algoritmo es definido en la Sección 5.1 de los Apéndices

solución parcial del problema y $c_{aux} = \sum_{j \in L^i} C_j$, sabemos que si $c_{aux} > M$ entonces no va a existir una posible extensión de L^i , pues los valores de contagio son números enteros positivos y por lo tanto no es posible que la suma de los contagios sea menor o igual a M . A través de esta poda podemos evitar ver el subárbol de la solución parcial de L^i y ahorrarnos operaciones innecesarias.

Poda por optimalidad Dado que nos interesa ofrecer como resultado de nuestro problema la combinación de locales que provea el mayor beneficio, podemos incluir una optimización en la resolución del mismo. En otras palabras, una vez que estamos tratando con *soluciones parciales* que consideramos *factibles*, podemos preguntarnos si hay una manera de saber si el beneficio de las futuras soluciones de las que forme parte nuestra solución parcial, ofrecerán un beneficio mayor al máximo beneficio *previamente* encontrado. De esta forma, se encuentra utilidad en incluir las variables B_{global} que almacenará el máximo beneficio encontrado hasta el momento por el algoritmo, y $B_{remanente}$, que se pasará por copia para cada recursión, y servirá para tener un registro de cuanto beneficio más se puede obtener frente a las decisiones tomadas hasta el momento, de esta manera, en caso de que la suma de b_{aux} (el beneficio hasta el momento) más $B_{remanente}$ sea menor o igual a B_{global} , no tendrá sentido continuar con futuras soluciones basadas en la actual, y se devuelve $-\infty$ para mostrar esto.

Algorithm 2 Algoritmo de Backtracking

```

1:  $B_{global} \leftarrow 0$ 
2: for  $0 \dots n - 1$  do
3:    $B_{remanente} \leftarrow B_{remanente} + B_i$ 
4:   function  $BT(n, B, C, M, i, b_{aux}, c_{aux}, B_{remanente})$ 
5:     if  $i \geq n$  then
6:       if  $c_{aux} \leq M$  then
7:          $B_{global} \leftarrow \max\{B_{global}, b_{aux}\}$ 
8:         return  $b_{aux}$ 
9:       else
10:        return  $-\infty$ 
11:     if  $c_{aux} > M$  then                                 $\triangleright$  Factibilidad
12:       return  $-\infty$ 
13:     if  $(b_{aux} + B_{remanente}) \leq B_{global}$  then       $\triangleright$  Optimalidad
14:       return  $-\infty$ 
15:     if  $i = n - 1$  then
16:        $val \leftarrow 0$ 
17:     else
18:        $val \leftarrow B_{i+1}$ 
19:     return  $\max\{BT(n, B, C, M, i + 1, b_{aux}, c_{aux}, B_{remanente} - B_i),$ 
20:              $BT(n, B, C, M, i + 2, b_{aux} + B_i, c_{aux} + C_i, B_{remanente} - B_i - val)\}$ 

```

La complejidad del algoritmo en peor caso es de $O(2^n)$. Esto se debe a que en el peor escenario, no se logra podar ninguna rama de las soluciones *posibles* y por lo tanto se termina generando el árbol con todas ellas, donde los nodos representarán el revisar el local contiguo si no tome el local sobre el que está trabajando el algoritmo, o el que le sigue, en caso de sí haberlo tomado. Si bien no se genera un árbol binario completo como en el caso de Fuerza Bruta, podemos notar que la rama con mayor longitud es aquella que representa la solución de no abrir ningún local (de longitud n), la cual es igual a la del árbol generado con Fuerza Bruta, como se aprecia en la Figura 1. Además sabemos que la longitud de las demás ramas serán menores a ella y mayores a la que representa la solución en la cual se agregan todos los locales posibles (distanciados entre si) cuya longitud es $\lceil \frac{n}{2} \rceil$. Por lo tanto, la cantidad de nodos del árbol generado por el algoritmo 2 es $O(2^n)$.

Adicionalmente, el código provisto muestra operaciones constantes en las líneas 1 y 5 a 18. Notar que aquellas soluciones donde $L = [l_1, \dots, l_j, i_h, \dots, l_n]$ tales que sus beneficios se encuentran ordenados de menor a mayor y que $b_j < \sum_{i=j+1}^n b_i$ para $1 \leq j \leq n$ con valores de contagio asociados tales que no excedan el límite de contagio, son las que requerirán generar todo el árbol de soluciones. Además, el mejor caso se dará cuando nos encontramos con que $\sum_{i=2}^n b_i = 0 \wedge 0 < b_1$, y $c_1 = M$ y $\forall j$ tal que $1 \leq j \leq n, c_j = M + 1$, de esta forma no tendría sentido corroborar las soluciones subsiguientes, y serían filtradas por ambas podas ya que nos encontramos en el límite de contagio y a su vez tenemos el máximo beneficio acumulable posible (notar que en este caso, el algoritmo pertenecerá a $O(n)$).

2.3. Programación Dinámica

Los algoritmos de **Programación Dinámica** son utilizados cuando un problema recursivo tiene solapamiento de subproblemas, es decir, para resolver el problema original, calculo un mismo resultado más de una vez. Para evitar esto definimos la siguiente función recursiva que soluciona dicho problema:

$$f(i, m) = \begin{cases} -\infty & m < 0 \\ 0 & i \geq n \\ \max(f(i + 2, m - C_i) + B_i, f(i + 1, m)) & \text{caso contrario} \end{cases}$$

Podemos definir $f(i, w)$ como el máximo beneficio producido por un subconjunto de L , llamado $L' = \{l_k, \dots, l_j\}$ tal que $w \leq W$. Notamos que $f(0, M)$ es el máximo beneficio producido dada una secuencia de locales L , cuyo nivel de contagio es menor o igual a M .

- (I) Si $m < 0$ entonces la solución parcial ya no sera una solución factible porque excede el límite de contagio, y tampoco lo serán otras que la contengan. Es por esto que devuelvo $-\infty$.
- (II) Si $i \geq n$ (cuyos casos posibles son $i = n$ e $i = n + 1$) entonces estamos diciendo que queremos encontrar un subconjunto del \emptyset tal que $m \leq M$ lo cual no es posible, por lo tanto vale 0.
- (III) En este caso, $i < n$ y $m \geq 0$, por consiguiente, buscamos el subconjunto de locales $L^i = \{l_j, \dots, l_k\}$ tales que $\sum_{h \in L^i} C_h \leq M$, o dicho de otra forma, $M - \sum_{h \in L^i} C_h \geq 0$. De existir un subconjunto, tiene que tener al i -ésimo local o no tenerlo. En caso de tenerlo, entonces tiene que ser a su vez un subconjunto de L^{i+2} que no supere el riesgo de contagio, como se nos pide que no abramos dos locales contiguos, entonces avanzamos al sucesor del contiguo. Entonces, debe llamarse de forma recursiva $f(i + 2, m - C_i) + B_i$. En caso contrario avanza de a un local recursivamente ($f(i + 1, m)$).

Memoización Se puede observar que la función dada para nuestro problema toma dos parámetros $i \in [0, \dots, n + 1]$ y $m \in [0, \dots, M]$ y que en los escenarios donde se presente el que $i \geq n$ o $m < 0$, son casos base y se pueden resolver de manera ad-hoc en tiempo constante.

De esta forma, las distintas combinaciones posibles para efectuar un llamado de nuestra función, están determinadas por la combinación de los parámetros de la misma. En este caso, hay $\Theta(n \times M)$ combinaciones posibles de parámetros.

Haciendo uso de esto, si agregamos una *memoria* que recuerde los resultados de llamados previamente efectuados de esta función, podemos calcular una sola vez cada uno de ellos y asegurarnos no resolver más de $\Theta(n \times M)$ casos (pues las situaciones donde se requiera saber el resultado de algo ya calculado, como el resultado fue almacenado, es utilizado como pre-cálculo en $O(1)$).

Esto último (el uso de la memoización) se puede observar en la línea 11, donde se efectua el llamado recursivo en caso de que la combinación de parámetros dada no hubiera sido calculada previamente.

Algorithm 3 Algoritmo de Programación Dinámica *top-down*

```

1: for  $i \in \{0 \dots n + 1\}$  do
2:   for  $j \in \{0 \dots M\}$  do
3:      $Mat[i, j] \leftarrow -1$ 
4:   function  $DP(i, m)$ 
5:     if  $m < 0$  then
6:       return  $-\infty$ 
7:     if  $Mat[i, m] == -1$  then
8:       if  $i \geq n$  then
9:          $Mat[i, m] \leftarrow 0$ 
10:      else
11:         $Mat[i, m] \leftarrow \max\{DP(i + 2, m - C_i) + B_i, DP(i + 1, m)\}$ 
12:   return  $Mat[i, m]$ 

```

La complejidad del algoritmo propuesto está determinada por la cantidad de llamados distintos de nuestra función, y el costo de su resolución. Como fue mencionado anteriormente, se resolverán a lo sumo $\Theta(n \times M)$ combinaciones de parámetros distintas, y dado que todas las líneas del algoritmo realizan operaciones constantes, cada llamado se resuelve en $O(1)$. Entonces, se deduce que la complejidad de este algoritmo pertenece a $O(n \times M)$ en el peor caso.

Se puede notar adicionalmente que la estructura de memoización elegida se puede implementar como una matriz con acceso y escritura constante. Es más, se puede observar que su inicialización tiene costo $\Theta(n \times M)$, por lo tanto, el mejor y peor caso del algoritmo que proponemos, pertenece a $\Theta(n \times M)$.

3. Experimentación

A continuación se encuentran los experimentos computacionales realizados para evaluar y comparar los algoritmos presentados en la Sección anterior. Las ejecuciones fueron realizadas en dos workstation con las siguientes especificaciones:

- CPU AMD® FX-8320e 3.2GHz y 8GB de memoria RAM,
 - CPU AMD® Ryzen 3 2200g 3.5GHz y 8GB de memoria RAM,
- y utilizando el lenguaje C++.

3.1. Métodos

Los métodos utilizados para la experimentación son:

- **FB**: Algoritmo 1 de Fuerza Bruta de la Sección 2.1
- **BT**: Algoritmo 2 de Backtracking de la Sección 2.2, con ambas podas.
- **BT-O**: Algoritmo 2 de Backtracking con la excepción de que cuenta solo con la poda de *optimalidad*, es decir, quitando las líneas 11 y 12.
- **BT-F**: Similar al método BT-O, pero aplicando únicamente la poda por factibilidad, es decir, descartando las líneas 13 y 14 del algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 2.3

3.2. Instancias

Los *datasets* definidos, son los siguientes²:

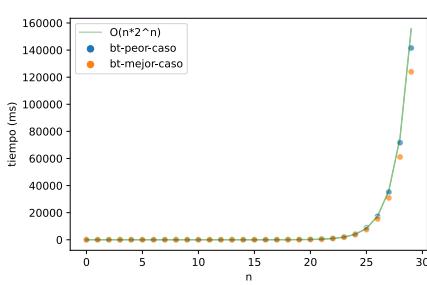
- **beneficio-bajo-contagio-alto**: Para todo elemento de B y C, $\forall i / 1 \leq i \leq n$ se tiene que $b_i \in N(0, \frac{M^2}{64})$, $c_i \in N(M, \frac{M^2}{64})$, y M fijo.
- **beneficio-alto-contagio-bajo**: Para todo elemento de B y C, $\forall i / 1 \leq i \leq n$ se tiene que $c_i \in N(0, \frac{M^2}{64})$ y $b_i \in N(M, \frac{M^2}{64})$, y M fijo.
- **bt-mejor-caso**: Para todo elemento de B se tiene que $b_i = 0 \wedge c_i = M + 1 \forall 1 < i \leq n$ y $0 < b_1$, y $c_1 = M$.
- **bt-peor-caso**: Para todo elemento de B se tiene que $b_j < \sum_{i=j+1}^n b_i$ para $1 \leq j \leq n$ y $c_i = 0, \forall i / 1 \leq i \leq n$.
- **dinamica**: Instancias para DP de valores para $n \in [50, 5000]$ con $M \in [50, n]$.
- **bt-vs-d-mejor**: Instancias para comparar Backtracking y Programación Dinámica cuando $M \geq 2^n$.

3.3. Experimento 1: Fuerza Bruta

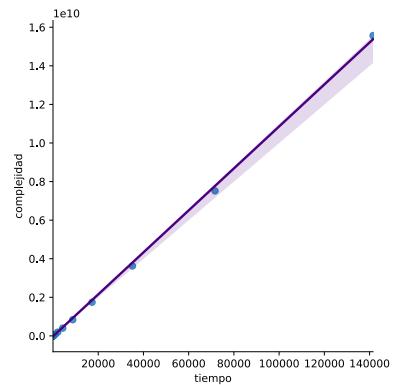
En este experimento se analiza el rendimiento del método FB para distintos conjuntos de instancias. Como hemos visto en la Sección 2, los tiempos de ejecución de mejor y peor caso de este método son iguales, ambos son exponenciales en función de n . Para demostrarlo empíricamente, se evalúa el algoritmo con las instancias de bt-mejor-caso y bt-peor-caso y se grafican los tiempos de ejecución en función de n .

Podemos ver en la Figura 2a como los tiempos de ejecución resultantes del experimento con las instancias de peor y mejor caso son bastante similares. Esto nos demuestra que el rendimiento de FB no se verá alterado según las características de las instancias, es decir, siempre tendrá un comportamiento exponencial. Además analizamos la correlación con la complejidad teórica del método, la cual es $O(n \times 2^n)$. Se puede visualizar como la

²Para todo valor de B y C se tomó su valor absoluto



(a) Tiempos de ejecución del algoritmo de Fuerza Bruta sobre las instancias bt-mejor-caso y bt-peor-caso contra la complejidad esperada.



(b) Correlación entre el tiempo de ejecución de FB y la cota de complejidad.

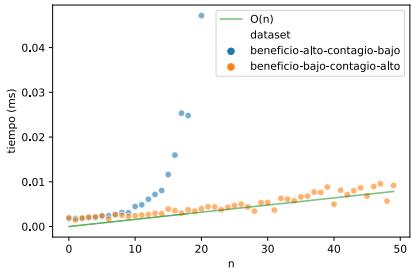
Figura 2: Análisis de complejidad del método FB

función exponencial se superpone perfectamente con los tiempos de ejecución de ambas instancias. En el gráfico de correlación entre la complejidad teórica y los tiempos de ejecución (Figura 2b) se aprecia como el tiempo de ejecución de cada instancia I_i de tamaño $i \in 1 \leq i \leq n$ cumple con el tiempo teórico analizado previamente. En particular, el índice de correlación de Pearson es $r \approx 0,999708$. Con esto obtuvimos una evidencia práctica de que el algoritmo se comporta como fue descripto previamente.

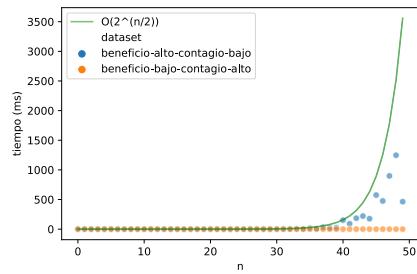
3.4. Experimento 2: Backtracking

Ahora vamos a analizar el comportamiento del método BT dadas instancias con variaciones de beneficios y contagios, y su complejidad. En este caso utilizamos los datasets beneficio-alto-contagio-bajo y beneficio-bajo-contagio-alto.

Las Figuras 3a y 3b grafican los tiempos de ejecución de BT para instancias de los datasets mencionados con $n \in 1 \leq n \leq 50$. En la de la derecha, se observa como para el dataset de beneficio alto y contagio bajo el algoritmo posee un comportamiento exponencial en función de n , aunque no supera a la cota teórica. Esta performance se debe a que al ser bajos los contagios respecto a M , la poda por factibilidad no se ejecuta la mayoría de las veces, por lo cual se genera una gran cantidad de nodos del árbol de recursión. En cambio, en la imagen de la izquierda, podemos observar que el comportamiento del algoritmo es lineal cuando el contagio es alto, ya que las soluciones parciales superan rápidamente el límite M . Para observar esto en detalle realizamos gráficos de correlación correspondientes a los tiempos de ejecución con las cotas: $O(2^n)$ para el dataset de contagio bajo y $O(n)$ para el de contagio alto.

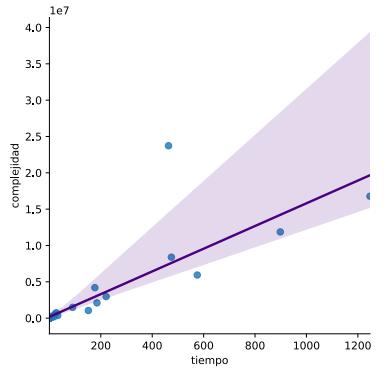


(a) Tiempo de ejecución del método BT sobre beneficio-alto-contagio-bajo y beneficio-bajo-contagio-alto, comparando este último con la complejidad esperada

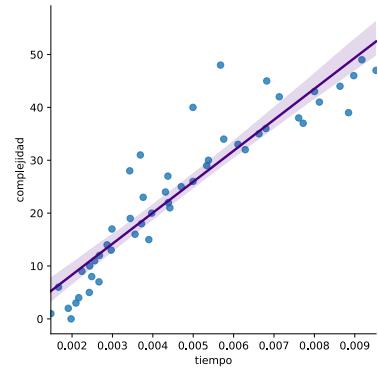


(b) Tiempo de ejecución del método BT sobre beneficio-alto-contagio-bajo y beneficio-bajo-contagio-alto, comparando el primero con la curva $2^{\frac{n}{2}}$

Figura 3: Análisis de complejidad del método BT



(a) Correlación entre el tiempo de ejecución de BT para el dataset beneficio-alto-contagio-bajo y la cota de complejidad.



(b) Correlación entre el tiempo de ejecución de BT para el dataset beneficio-bajo-contagio-alto y la cota de complejidad.

Figura 4: Análisis de correlación del método BT

Para las instancias de contagio bajo comparamos los tiempos de ejecución con la curva $2^{\frac{n}{2}}$ puesto que al realizar el análisis con la cota de complejidad exponencial propuesta en la Sección 2.2 notamos que el índice de correlación era no muy alto ($r \approx 0,767175$), en cambio con la nueva es $r \approx 0,893267$. De igual forma, ambas funciones pertenecen a $O(2^n)$, por lo tanto, concluimos que la performance del algoritmo para este dataset es exponencial. De la misma forma, podemos ver como en la Figura 4b, si bien hay variaciones, hay una correlación bastante alta entre los tiempos de ejecución y la cota lineal, más específicamente, el índice de correlación de Pearson es $r \approx 0,927736$.

Notamos que la variación en los beneficios no fue un factor determinante en estos casos en cuanto a los tiempos de ejecución. En el siguiente experimento veremos distintos datasets en los cuales el beneficio puede llegar a jugar un rol importante sobre la performance del algoritmo.

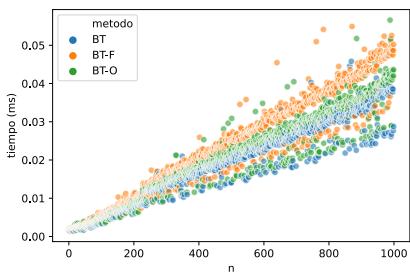
3.5. Experimento 3: Efectividad de las podas

Dentro de los escenarios posibles que se pueden dar al ejecutar el algoritmo, es interesante poder entender qué partes del mismo proveen un mejor rendimiento, dependiendo de la información con la que se trabaje.

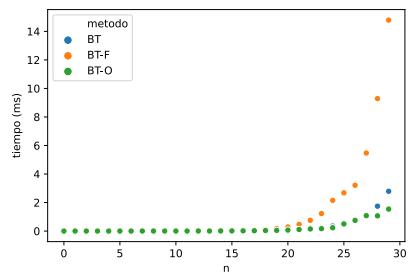
Acá es donde entran en juego las *podas* utilizadas al diseñar el algoritmo. Por ejemplo, una hipótesis es que el uso de la *poda por factibilidad* permitirá obtener un comportamiento lineal si los valores de contagio de los locales exceden el límite de contagio.

Por otro lado, la *poda por optimidad* se puede aprovechar cuando el beneficio del primer elemento es mayor que la suma de todos los elementos restantes, o más aún, si los demás elementos no tienen beneficio, se obtiene un comportamiento lineal.

En este experimento se compara el funcionamiento de los métodos BT, BT-F, BT-O con respecto a los datasets bt-mejor-caso y bt-peor-caso. La hipótesis es que (ilustrando las situaciones descriptas arriba) las podas lograrán reducir la complejidad del algoritmo, de modo de que al necesitar o utilizar menos elementos, el mismo sea más eficiente.



(a) Tiempos de ejecución de los métodos BT, BT-O y BT-F sobre el conjunto de instancias bt-mejor-caso.



(b) Tiempos de ejecución de los métodos BT, BT-O y BT-F sobre el conjunto de instancias bt-peor-caso.

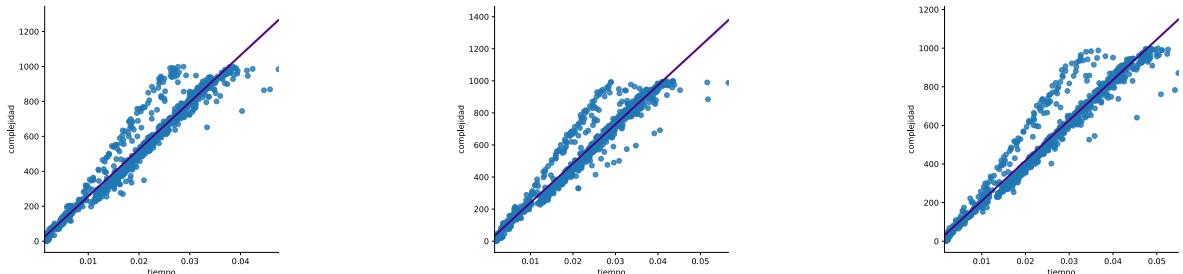
Figura 5: Análisis de los métodos BT,BT-O y BT-F

En la Figura 5 se pueden observar los resultados de utilizar los métodos BT,BT-O y BT-F sobre los datasets *bt-mejor-caso* y *bt-peor-caso*.

Un detalle a notar es que la ejecución sobre cada dataset difiere en su n , pues la complejidad del peor caso de Backtracking resulta exponencial, con lo que se tomó decisión de utilizar $n = 30$ para tal escenario, y $n = 1000$ para el mejor caso.

Por un lado, en la Figura 5a no hay una distinción tan marcada sobre la utilización de ambas podas o una sola, aunque BT-O pareciera operar más eficientemente que BT-F. Por el otro, esto sí es fácil de observar en la Figura 5b; lo cual puede deberse a que es más eficiente definir el posible beneficio *remanente* a obtener de los locales restantes, que verificar que el contagio acumulado no exceda el límite, pues esto puede llegar a efectuarse en cada solución encontrada inclusive.

En la Figura 6 se tienen la correlación entre las ejecuciones y una cota de complejidad lineal. Siendo estos coeficientes $r \approx 0,967832$ para 6a, $r \approx 0,965495$ para 6b y $r \approx 0,971311$ para 6c.



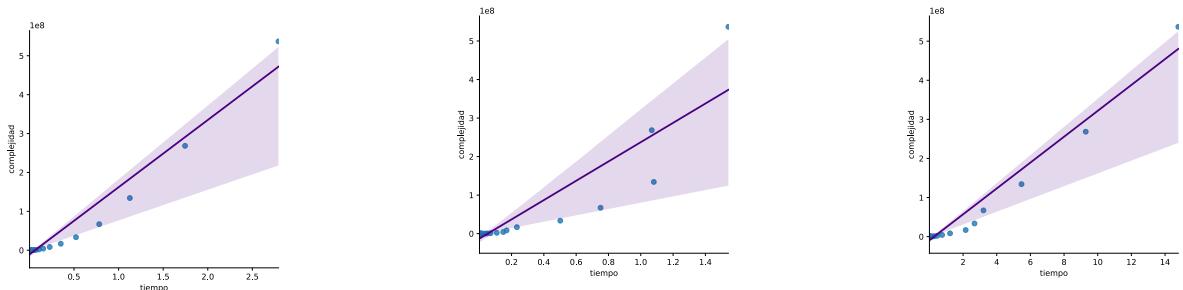
(a) Correlación entre el tiempo de ejecución de BT para el dataset mejor-caso y la cota de complejidad.

(b) Correlación entre el tiempo de ejecución de BT-O para el dataset mejor-caso y la cota de complejidad.

(c) Correlación entre el tiempo de ejecución de BT-F para el dataset mejor-caso y la cota de complejidad.

Figura 6: Análisis de efectividad de las podas de BT-1

En la Figura 7 se tienen la correlación entre las ejecuciones y una cota de complejidad lineal. Siendo estos coeficientes $r \approx 0,976571$ para 7a, $r \approx 0,902164$ para 7b y $r \approx 0,981454$ para 7c.



(a) Correlación entre el tiempo de ejecución de BT para el dataset bt-peor-caso y la cota de complejidad.

(b) Correlación entre el tiempo de ejecución de BT-O para el dataset bt-peor-caso y la cota de complejidad.

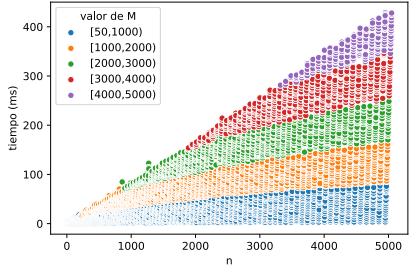
(c) Correlación entre el tiempo de ejecución de BT-F para el dataset bt-peor-caso y la cota de complejidad.

Figura 7: Análisis de efectividad de las podas de BT-2

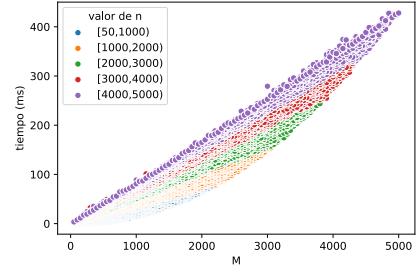
Las diversas ejecuciones corroboran las cotas de complejidad previamente desarrolladas.

3.6. Experimento 4: Programación Dinámica

A continuación, se estudia la eficiencia del algoritmo de Programación Dinámica en la práctica y su correlación con la cota teórica analizada en la Sección 2.3. Para ello, se ejecutan las instancias del dataset dinámica sobre el método DP y se grafican sus resultados.



(a) Tiempo de ejecución del método DP en función de n para el dataset dinámica



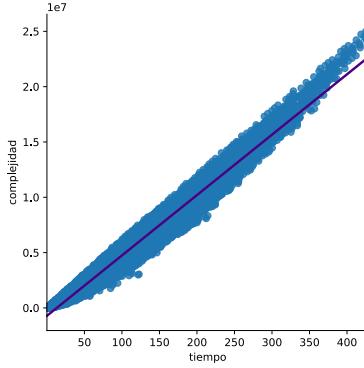
(b) Tiempo de ejecución del método DP en función de M para el dataset dinámica

Figura 8: Análisis de complejidad del método DP

En la Figura 8 se puede observar el crecimiento del tiempo de ejecución del algoritmo en función de n y M . Se decidió realizar cortes en valores particulares de n y M para facilitar el análisis.

En la figura 8a se observa la ejecución en función de n distinguiendo algunos valores de M por color, y en la Figura 8b la ejecución en función de M para distintos valores de n . Ambas figuras muestran un comportamiento lineal en función de su respectiva variable.

Por otro lado, adicionalmente, verificamos su coeficiente de correlación con la cota teórica (el cual se puede ver en la Figura 9a) siendo este $r \approx 0,986567$.



(a) Correlación entre el tiempo de ejecución de DP para el dataset dinámica y la cota de complejidad.

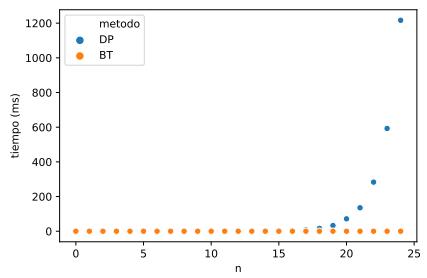
Figura 9: Análisis de correlación del método DP

3.7. Experimento 5: Backtracking vs Programación Dinámica

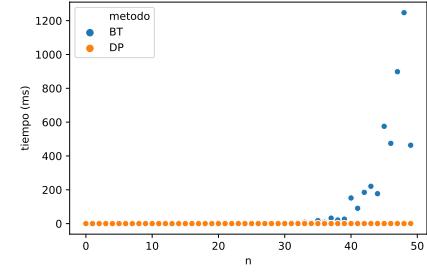
Por último, consideramos que es útil comparar dos técnicas algorítmicas. Con ello, es posible obtener más información que permita determinar cuando es conveniente el uso de una u otra, dependiendo de la instancia dada. La hipótesis con la que queremos realizar este análisis, es que la utilidad de cada técnica provendrá de instancias distintas.

Esto puede verse al utilizar una estructura de memoización de mantenimiento costoso al encontrarse con instancias donde la cantidad de locales necesarios a revisar y tener en cuenta para llegar a la solución óptima es baja, en contraste con el uso eficiente de las podas para tal instancia; aunque por otro lado, si dicha cantidad es alta, el uso de una técnica de Backtracking puede tener un costo temporal elevado, mientras que obtener una solución por Programación Dinámica resulte mucho más eficiente.

Algo relevante que se puede notar, es que no se puede establecer una cota de complejidad de una técnica por sobre la otra, dado que $O(2^n) \not\subseteq O(n \times M)$ y a su vez $O(n \times M) \not\subseteq O(2^n)$.



(a) BT vs DP para el dataset bt-vs-d-mejor



(b) BT vs DP para el dataset beneficio-alto-contagio-bajo

Figura 10: Comparación de ejecución entre DP y BT

Observando la complejidad de ambos métodos, se puede notar que el tiempo de ejecución de BT no depende del valor de M , pues depende únicamente de n . Pero esto no es así con el método DP, si $M = 2^n$, la complejidad $O(n \times M)$ pasaría a ser $O(n \times 2^n)$ la cual excedería la complejidad de BT.

La Figura 10 muestra la comparación de estos métodos sobre los dataset bt-vs-d-mejor y beneficio-alto-contagio-bajo. La hipótesis se confirma al ver que en los casos donde el M es bajo, DP hace un uso de la memoización y resulta en una ejecución más eficiente (Figura 10b), pero por otro lado cuando $M \approx 2^n$ se llega a un uso extensivo de memoria y resulta en una ejecución ineficiente para DP (Figura 10a).

4. Conclusiones

En este trabajo se presentan tres algoritmos que utilizan técnicas distintas para resolver el problema del distanciamiento social.

El algoritmo de Fuerza Bruta resulta no ser muy eficiente para resolver este problema dado que al aumentar la cantidad de locales de L crece su tiempo de ejecución de forma exponencial. El algoritmo de Backtracking provee una mejora a través de sus podas, las cuales demuestran ser de utilidad en la mayoría de las instancias; logran bajar el crecimiento de los tiempos de ejecución cuando las instancias poseen determinada estructura (véase el análisis de mejor caso). Por último, el algoritmo de Programación Dinámica es el más robusto frente al crecimiento de la cantidad de locales n , aunque este se ve afectado de forma directa por el tamaño de M , lo que hace que frente valores de M muy grandes no sea la mejor elección.

5. Apéndice

5.1. Función *distanciados*

El algoritmo siguiente es el utilizado en el algoritmo de Fuerza Bruta para corroborar la factibilidad de la solución encontrada, analizando que los locales no sean contiguos.

Algorithm 4 función para chequear distancia

```

1: function distanciados( $A$ )
2:   if  $|A| == 0$  then
3:     return false
4:   if  $|A| == 1$  then
5:     return true
6:    $cumple \leftarrow true$ 
7:   for  $0 \dots |A| - 1$  do
8:     if  $A_i + 1 == A_{i+1}$  then
9:        $cumple \leftarrow false$ 
10:  return cumple

```
