



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Técnicas algorítmicas con distanciamiento social

Algoritmos y Estructuras de Datos 3

Integrante	LU	Correo electrónico
Gómez, Bruno Agustín	428/18	bgomez@dc.uba.ar



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Introducción	3
2. Desarrollo	3
2.1. Fuerza Bruta (FB)	3
2.2. Backtracking (BT)	4
2.3. Programación Dinámica (PD)	5
3. Experimentación	6
3.1. Métodos	6
3.2. Instancias	6
3.3. Experimento 1: Complejidad de Fuerza Bruta	7
3.4. Experimento 2: Complejidad de Backtracking	8
3.5. Experimento 3: Efectividad de las podas	9
3.6. Experimento 4: Complejidad de Programación Dinámica	10
3.7. Experimento 5: Backtracking vs Programación Dinámica	11
4. Conclusiones y trabajo futuro	11
5. Apéndice	12
5.1. Demostración de la complejidad de Fuerza Bruta	12
5.2. Influencia de los resultados en el Experimento 1 con el Experimento 2	12

1. Introducción

El problema *negocio por medio* (NPM) que tratamos en este trabajo práctico es un problema de optimización combinatoria, es decir, busca encontrar la solución óptima entre un conjunto finito de soluciones posibles. Formalmente, se trata de una secuencia de n locales ordenados $L = [l_1, \dots, l_n]$, donde cada local tiene como características un beneficio y un valor de contagio $b_i, c_i \in \mathbb{N}_{\geq 0}$. Contamos, también, con un límite de valor de contagio $M \in \mathbb{N}_{\geq 0}$ que no puede ser superado por la suma total del valor de contagio de los locales que están abiertos. El conjunto de soluciones de NPM para un conjunto de locales L se define como:

$$\{L' \subseteq L \wedge l_i \in L' \wedge (l_{i+1} \notin L' \wedge l_{i+1} \notin L')\}$$

Por otro lado, una solución L' , para ser factible, debe cumplir que $\sum_{l_i \in L'} c_i \leq M$, donde c_i es el contagio correspondiente al local l_i .

A continuación, vamos a mostrar como ejemplo distintas instancias del problema junto con sus soluciones. Si $M = 20$, $L = \{1, 2, 3, 4\}$ y los locales tienen las siguientes características:



Figura 1: Características de los locales de nuestro ejemplo.

Nos encontramos que las soluciones factibles son $L_1 = \{l_1, l_3\}$, $L_2 = \{l_1, l_4\}$, $L_3 = \{l_1\}$, $L_4 = \{l_3\}$, $L_5 = \{l_4\}$, siendo L_2 la más óptima. Las soluciones que no son factibles serían $\{l_2\}, \{l_2, l_4\}$ ya que superaría el límite de contagio. Por otro lado, si $M = 4$, $L = \{1, 2, 3, 4\}$ y los locales son los ya detallados en la Figura 1, no existe ninguna solución factible, debido a que ningún local tiene un valor de contagio inferior al límite, por lo que este se vería siempre superado. Notar que mientras exista un local con un valor de contagio menor al límite, existe al menos una solución factible.

El enfoque de este trabajo práctico va a estar dirigido a resolver NPM con distintas estrategias para luego realizar una comparación entre las diferentes soluciones. En primera instancia, vamos a intentar resolverlo usando la técnica algorítmica de *Fuerza Bruta* (BT), que consiste en enumerar recursivamente todas las posibles soluciones, quedándonos sólo con aquellas que sean factibles. Considerando que la estrategia anterior tiene un costo computacional potencialmente prohibitivo, se aprovechan las propiedades del problema para evitar analizar todas las soluciones posibles. A esta reducción de elementos explorados en el árbol de búsqueda se le llama *poda* y da lugar a nuestra segunda estrategia llamada *Backtracking* (BT). Finalmente, considerando el solapamiento de subproblemas que puede ocurrir en los llamados recursivos para evitar resolver más de una vez la misma instancia, introducimos la técnica de memoización, dando lugar a una última técnica denominada *Programación Dinámica* (DP).

2. Desarrollo

2.1. Fuerza Bruta (FB)

Cómo mencionamos anteriormente, un algoritmo de Fuerza Bruta enumera todas las posibles soluciones para luego seleccionar aquellas válidas o las más óptimas. En nuestro caso, el conjunto de soluciones está compuesto por todos los subconjuntos de L tales que no contengan elementos adyacentes. Las soluciones factibles están dadas por:

$$S = \{s | s \in \mathcal{P}(L) \wedge \sum_{l_i \in s} c_i \leq M \wedge l_i \in s \implies (l_{i-1} \notin s \wedge l_{i+1} \notin s)\}$$

Sea $L = \{l_1, \dots, l_n\}$ y M el límite de contagio, tal que $l_i = (b_i, c_i)$ donde b_i es el beneficio y c_i es el contagio asociado al iésimo local. Se denota al conjunto de partes de L como $\mathcal{P}(L)$.

Por ejemplo, si $L = (10,20),(30,10),(15,15), (|L| = n)$ y $M = 30$, el conjunto de soluciones sería $\{(10,20), (30,10), (15,15), ((10,20),(30,10))\}$ y el conjunto de soluciones factibles $S = \{(10,20), (30,10), (15,15)\}$.

En líneas generales, el algoritmo de Fuerza Bruta 1 genera todas las soluciones decidiendo en cada paso si el iésimo local es seleccionado o no, y quedándose con la selección de locales que le otorga el máximo beneficio de alguna de esas dos ramas. Para no seleccionar locales adyacentes, si se selecciona el local actual, la recursión continúa con el $(i+2)$ -ésimo, en el caso contrario se continúa con el $(i+1)$ -ésimo. Por último, cuando tenemos la solución, determinaremos la factibilidad de la solución si el contagio acumulado es menor que el límite de contagio establecido, de ser así devolvemos el beneficio obtenido para esa solución.

Algorithm 1 Algoritmo de Fuerza Bruta

```

1: function NPM_FB( $i, contagio$ )
2:   Caso Base
3:   if  $i \geq n$  then
4:     if  $contagio \leq M$  then retorno 0 else retorno  $-\infty$                                  $\triangleright O(1)$ 
5:   Recursión
6:   retorno max(NPM_FB( $NPM_{FB}(i+2, contagio + c_i) + b_i, NPM_{FB}(i+1, contagio)$ ))       $\triangleright T(n-1) + T(n-2)$ 

```

Donde c_i y b_i denotan el contagio y el beneficio del iésimo local.

La correctitud del algoritmo 1 se puede argumentar basándose en que se generan todas las posibles soluciones para el conjunto de locales dado, donde para cada local, se crean dos posibles soluciones: una que lo contiene en el conjunto y la otra no. Sabiendo que se generan todas las posibles soluciones, de existir, debe encontrarse la solución óptima. Como se puede ver en la Figura 2, cada nodo intermedio representa una *solución parcial*, es decir, cuando aún no se decidió sobre todos los locales a incluir. Por otro lado, las hojas representan todas las soluciones.

La complejidad del algoritmo de Fuerza Bruta 1 para el peor caso es $\mathcal{O}(2^n)$. Intuitivamente, se puede ver que el árbol recursivo que se genera en nuestro algoritmo estaría contenido en el árbol que explora todas las soluciones, hasta las adyacentes. Esto es demostrado rigurosamente por el resultado obtenido en el apéndice[5.1]. Es importante señalar que la solución de cada llamado recursivo es de tiempo constante como se puede observar en las líneas 3, 4 y 6; solamente se hacen operaciones elementales como suma, comparaciones y máximo. Además, por el resultado[5.1], se puede ver que su mejor caso es de menor complejidad que el peor caso.

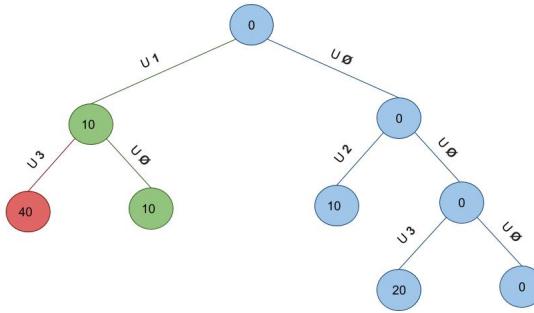


Figura 2: Representación gráfica del árbol de recursión del algoritmo, usando como valores los especificados en el ejemplo descrito en la introducción.

Árbol de recursión generado para la instancia $L = (10,20),(30,10),(15,15)$ y $M = 30$, donde cada nodo contiene el número de contagio obtenido. En verde la solución óptima, en azul las soluciones factibles y en rojo las soluciones negativas.

2.2. Backtracking (BT)

Los algoritmos de Backtracking son una modificación de la técnica de Fuerza Bruta que aprovechan propiedades del problema para evitar analizar todas las soluciones posibles. La idea es enumerar todas las soluciones formando el *árbol de backtracking*, donde en cada nodo se genera una rama por cada decisión y se mantiene la mejor solución hallada en alguna de estas. Para evitar explorar todo el conjunto de soluciones, se utilizan las *podas*, reglas que nos evitan continuar la exploración por ramas que no nos van a llevar a la solución deseada. Las podas en este caso van a ser por *factibilidad* y *optimalidad*.

Poda por factibilidad: Esta poda aprovecha el límite de contagio que no debe ser superado por los locales seleccionados. Sea L' la solución parcial representada por determinado nodo n_i con $m = \sum_{l \in L'} c_l$ la suma de los contagios obtenidos hasta el momento, dado que todos los contagios son números positivos, si $m > M$, entonces no podríamos extender o mantener la solución parcial L' para que la suma de sus contagios sea menor o igual al límite establecido M . Con esta regla, podemos evitar seguir explorando el subárbol por debajo de n_i , reduciendo la cantidad de soluciones visitadas. Esta poda se encuentra en la línea 9 del algoritmo 2.

Poda por optimalidad: Supongamos que tenemos una solución factible con beneficio B . Sea L' una solución parcial en el nodo n_i con beneficio b y sea r la suma de todos los beneficios de cada local que falta decidir, si ocurre que $b + r < B$, cualquier decisión que se tome en el subárbol formado por debajo de n_i va a lograr un beneficio $b + r$ como mucho. Podemos asegurar, entonces, que cualquier solución factible que se pueda encontrar por debajo de n_i va a ser peor que la que ya conocemos. Esto nos permite podar esta rama y evitar el cómputo de instancias innecesarias. En el algoritmo 2 se actualiza una variable global B cada vez que se haya una solución factible en la línea 5, y se aplica la poda en la línea 12.

Algorithm 2 Algoritmo de Backtracking

```

1: function  $NPM_{BT}(i, contagio, beneficio)$ 
2:   Caso Base
3:   if  $i \geq n$  then
4:     if  $contagio \leq M$  then ▷ O(1)
5:        $B \leftarrow \max\{B, beneficio\}$  ▷ O(1)
6:       retorno 0 ▷ O(1)
7:     else retorno  $-\infty$  ▷ O(1)
8:   if  $contagio \leq M$  then retorno  $-\infty$  ▷ Poda por factibilidad
9:    $r \leftarrow beneficio, i \leftarrow 0$ 
10:  while  $i < n$  do ▷ O(n)
11:     $r \leftarrow r + b_i$ 
12:     $i \leftarrow i+1$ 
13:    if  $r + beneficio < B$  then retorno  $-\infty$  ▷ Poda por optimalidad
14:    Recursión
15:  retorno  $\max(NPM_{BT}(i+2, contagio + c_i) + b_i, NPM_{BT}(i+1, contagio))$  ▷ T(n-1) + T(n-2)

```

La complejidad del algoritmo 2 en peor caso es $\mathcal{O}(n2^n)$. Esto se debe a que en el peor de los casos, las podas no reducen la cantidad de soluciones visitadas. Es decir, se enumeran las soluciones de la misma manera que en Fuerza Bruta pero agregando en cada decisión local el costo de la poda de optimalidad, el cual es proporcional a la cantidad de locales. Además, el código introducido en las líneas 6, 7, 8 y 10 solo genera un número constante de operaciones elementales. Notar que existen distintas familias de instancias para las cuales el algoritmo 2 se comporta de distinta manera. Por ejemplo, si $L = \{(1,1), (1,1), \dots, (B,M)\}$ ($|L|=n$) $M = n$ y $B = 2n$, entonces el algoritmo va a enumerar todo el árbol dado que la poda de factibilidad nunca va a lograr la condición para activarse y como la mejor solución se va a encontrar en la última rama la poda de optimalidad nunca va a podar ninguna rama. Por otro lado, si $L = \{(1,M+1), \dots, (1,M+1), (1,M+1)\}$ ($|L|=n$) con $M > 0$ se puede observar un comportamiento lineal, gracias a la poda de factibilidad que nos va a asegurar que no se van a enumerar más de $\mathcal{O}(n)$ nodos. Observemos que sea L tal que sus elementos cumplen que $l_k = (2,1)$ si k es par o $l_k = (0,1)$ si k es impar ($|L|=n$) con $M > 0$, se obtiene una solución óptima en la primer rama y luego por la poda de optimalidad se garantiza que ningún otro nodo se va a ramificar. Como se van a generar $\mathcal{O}(n)$ nodos y por cada uno el costo para la poda de optimalidad es de orden lineal, el comportamiento resultante para esta familia de instancias es de orden cuadrático $\mathcal{O}(n^2)$.

2.3. Programación Dinámica (PD)

Esta técnica algorítmica se caracteriza por evitar repetir llamadas recursivas almacenando los resultados que ya han sido calculados. Cuando el problema tiene superposición de subproblemas, genera una mejora significativa en la complejidad evitando recalcular el subárbol. Sin embargo, para que esta técnica sea aplicable, el problema debe cumplir el Principio de Optimalidad de Bellman¹. En nuestro caso, definimos la siguiente función recursiva para resolver el problema:

$$NPM(i, c) = \begin{cases} -\infty & \text{si } c > M \\ 0 & \text{si } i \geq n \wedge c \leq M \\ \max NPM(i+2, c + c_i) + b_i, NPM(i+1, c) & \text{caso contrario.} \end{cases}$$

Intuitivamente queremos decir que $NPM(i, c)$: ^{el} máximo beneficio de un subconjunto de $\{l_i, \dots, l_n\}$ no adyacentes tal que la suma de su valor de contagio es menor o igual que $M - c$ ". Además, se puede ver que $NPM(0,0)$ es ^{el} máximo beneficio que se obtiene por un subconjunto L tal que la suma de sus contagios sea menor o igual a M ". Demostremos que esta función resuelve correctamente nuestro problema.

Correctitud de la función de recursión

- (I) Si $c > M$, ocurre que ninguno de los subconjuntos que pueda formar va a poder lograr que $c - M < 0$, dado que todos los valores de contagio son números enteros positivos. Considerando esto la respuesta es $-\infty$.
- (II) Si $i \geq n \wedge c \leq M$, estamos buscando los subconjuntos de \emptyset tales que la suma de sus valores de contagio no supere $M - c$. Como en \emptyset no hay ningún elemento y, además, tenemos que $c \leq M$, tomamos el subconjunto vacío que suma cero al valor de contagio. Siguiendo este razonamiento, la respuesta es $NPM(i,c) = 0$.
- (III) Si tenemos que $i < n$ y $c \leq M$, nos encontramos buscando el subconjunto de $L^k = \{l_k, \dots, l_n\}$ tal que la suma de sus valores de contagio no supere $c' = M - c$ y tal que no contenga elementos adyacentes. Si existe dicho subconjunto, entonces puede contener o no al elemento l_k . Si no lo tiene, entonces el subconjunto tiene que ser subconjunto de L^{k+1} , no sumar más de c' como valor de contagio y no contener elementos adyacentes; esto quiere decir que debe encontrarse con el llamado recursivo $NPM(i+1, c)$. Por otro lado, si contiene al elemento l_k , entonces los elementos restantes no deben sumar más de $c' - c_k$ como valor de contagio, deben estar en L^{k+2} dado que no puede contener a l_{k+1} porque es adyacente a l_k , es decir, podemos conseguir la solución con el llamado recursivo $NPM(i+2, c + c_k) + b_k$. Entre todas estas soluciones debemos elegir la que

¹Dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima.

nos proporciona el mayor beneficio, por lo que la mejor solución sería $\text{NPM}(i, c) = \max\{\text{NPM}(i+1, c), \text{NPM}(i+2, c+c_k) + b_k\}$. Notar que al término de la derecha se le suma b_k por haber agregado a l_k .

Memoización: Notemos que la función recursiva[2.3] toma dos parámetros $i \in \{0, \dots, n-1\}$ y $c \in \{0, \dots, M\}$. Como $i \geq n$ o $c > M$ son mi caso base y se pueden resolver con un número constante de operaciones elementales en tiempo constante, la cantidad de estados con los que se puede llamar al algoritmo 3, está determinada por la combinación de los parámetros. En este caso, hay $\Theta(n * M)$ combinaciones posibles de parámetros. Teniendo en cuenta esto, si utilizamos una estructura que pueda almacenar cada estado resuelto y su correspondiente resultado, podemos calcular una sola vez cada uno de ellos y asegurarnos no resolver más de $\Theta(n * M)$ casos. El algoritmo 3 muestra esta idea aplicada a la función[2.3]. En la línea 8 se lleva a cabo el paso de memoización que solamente se ejecuta si el estado no había sido previamente computado.

Algorithm 3 Algoritmo de Programación Dinámica

```

1:  $\text{MEMO}_{j,m} \leftarrow \perp$  para  $j \in \{0, \dots, n-1\}$ ,  $m \in \{0, \dots, M\}$ 
2: function  $\text{NPM}_{DP}(i, c)$ 
3:   Caso Base
4:   if  $i \geq n \vee c > M$  then
5:     if  $c \leq M$  then retorno 0 else retorno  $-\infty$ 
6:   Si no fue computado, lo computo recursivamente
7:   if  $\text{MEMO}_{i,c} = \perp$  then
8:      $\text{MEMO}_{i,c} \leftarrow \max(\text{NPM}_{DP}(i+2, \text{contagio} + c_i) + b_i, \text{NPM}_{DP}(i+1, \text{contagio}))$   $\triangleright T(n-1) + T(n-2)$ 
9:   retorno  $\text{MEMO}_{i,c}$ 

```

La complejidad del algoritmo entonces está determinada por la cantidad de estados que se resuelven y el costo de resolver cada uno de ellos. Como mencionamos anteriormente, a lo sumo se resuelven $\mathcal{O}(n^*M)$ estados distintos, y como las líneas 4, 5, 7, 8, y 9 del algoritmo 3 realizan un número constante de operaciones elementales entonces cada estado se resuelve en $\mathcal{O}(1)$, por lo que puedo concluir que el algoritmo tiene complejidad $\mathcal{O}(n^*M)$ en el peor caso. Es importante observar que el diccionario **MEMO** se puede implementar como una matriz con acceso y escritura constante. Más aún, notar que su inicialización tiene costo $\Theta(n^*M)$, por lo tanto, el mejor y peor caso de nuestro algoritmo va a tener costo $\Theta(n^*M)$.

3. Experimentación

En esta sección se van a presentar los experimentos computacionales realizados para analizar los distintos métodos de resolución para el problema NPM, presentados en las secciones anteriores. La implementación de los algoritmos fue realizada en *C++* y el código para la experimentación en *Python*. La experimentación fue realizada en una PC con **CPU:AMD Ryzen 5 1600 Six-Core Processor**, con **memoria principal :8GB RAM DIMM DDR4 Synchronous Unbuffered @ 2400 MHz** y **memoria secundaria :1TB ATA Disk WDC WD10EZEX-75W @**.

3.1. Métodos

Las métodos y sus variaciones utilizados durante la experimentación son los siguientes:

- **FB:** Algoritmo 1 de Fuerza Bruta de la Sección 2.1.
- **BT:** Algoritmo 2 de Backtracking de la Sección 2.2.
- **BT-F:** Algoritmo 2 con excepción de la línea 14, es decir, solamente aplicando podas por factibilidad.
- **BT-O:** Similar al método BT-F pero solamente aplicando la poda por optimidad, es decir, descartando la línea 9 del Algoritmo 2.
- **DP:** Algoritmo 3 de Programación Dinámica de la Sección 2.3.

3.2. Instancias

Para analizar el desempeño los algoritmos en distintos escenarios es menester definir familias de instancias conformadas con distintas características. Por ejemplo, el algoritmo de Backtracking como se menciona en la Sección 2.2 tiene familias que producen mejores y peores casos para el algoritmo. Primero, antes de enumerar los *Datasets* vamos a introducir ciertas definiciones. Se define como *densidad de contagio* de una instancia como el cociente $\frac{\max c_i}{\max b_i}$, es decir, la proporción de contagio que aporta por beneficio. A menor *densidad de contagio*, los beneficios de los locales en L son más grandes en relación al contagio. Considerando esto vamos a mostrar como se definen los *Datasets* a continuación.

- **alto-contagio:** Esta familia se define con alta densidad de contagio, teniendo una cantidad de locales $n \in \{1, \dots, 150\}$ y $M = n^2$. Cada local se compone con un b que toma un valor aleatorio entre $[1, (n+1) * 2]$ y un c con un valor aleatorio entre $[n^2, (n+1) * 30]$. Si bien se tomaron intencionalmente los intervalos para que hayan casos donde b sea mayor que c intuitivamente se ve que en la mayoría de los casos esto no va a ocurrir. Para disminuir el “ruido” generado por valores aleatorios particulares, generamos 10 instancias con la misma cantidad de locales.

- **bajo-contagio:** Este conjunto de instancias se genera con una cantidad de locales $n \in \{1, \dots, 150\}$ y un límite de contagio $M = \max\{1, \lfloor \frac{n}{4} \rfloor\}$, donde predomina la baja densidad de contagio. Para cada local su beneficio b y contagio c toman valores aleatorios entre $[n+1, (n+1)*10]$ y $[0, \lfloor \frac{M}{2} \rfloor]$ respectivamente. Al igual que en el Dataset de **alto-contagio**, para disminuir el “ruido” generado por el sistema para cada n generamos 10 instancias.
- **bt-mejor-caso:** Para esta familia de instancias variamos la cantidad de locales $\in \{1, \dots, 1000\}$ y $M = n$. Cada instancia de n locales, está formada por $L = \{(1, M+1), \dots, (1, M+1)\}$. Son las instancias para el mejor caso de Backtracking definidas en la Sección 2.2.
- **bt-caso-intermedio:** En este conjunto de instancias se utiliza una cantidad de locales $n \in \{1, \dots, 150\}$ y $M = n$. Cada instancia de n locales, en su i -ésimo local si i es par $l_i = (2, 1)$ sino es $(0, 1)$. Estas instancias fueron definidas anteriormente en la Sección 2.2.
- **bt-peor-caso:** Cada instancia de n elementos, está formada por $L = \{(1, 1), \dots, (1, 1), (2n, M)\}$ y $M = n$. Son las instancias para el peor caso de Backtracking definidas en la Sección 2.2. La cantidad de locales n varía en el intervalo $[1, \dots, 45]$
- **variacion-n-m:** Esta familia de instancias tiene instancias con distintas combinaciones de valores para n y M en los intervalos $[100, 1700]$. Los beneficios y contagios para cada local poseen un valor aleatorio en el intervalo $[1, \dots, n * 10]$.

3.3. Experimento 1: Complejidad de Fuerza Bruta

Hipótesis 1. La complejidad encontrada empíricamente va a ser del orden K^n donde $\sqrt{2} \leq K \leq 2$

En este experimento se analiza el desempeño del método FB en distintos contextos. El análisis de complejidad realizado en la Sección 2.1 y fundamentado de manera rigurosa en la Sección 5.1 indica que el tiempo de ejecución es de orden exponencial en función de n pero se encuentra entre $\Omega(2^{\frac{n}{2}})$ y $\mathcal{O}(2^n)$. Para contrastar empíricamente estas afirmaciones se evalúa FB utilizando los Datasets **alto-contagio** y **bajo-contagio** y se grafica los tiempos de ejecución en función de n .

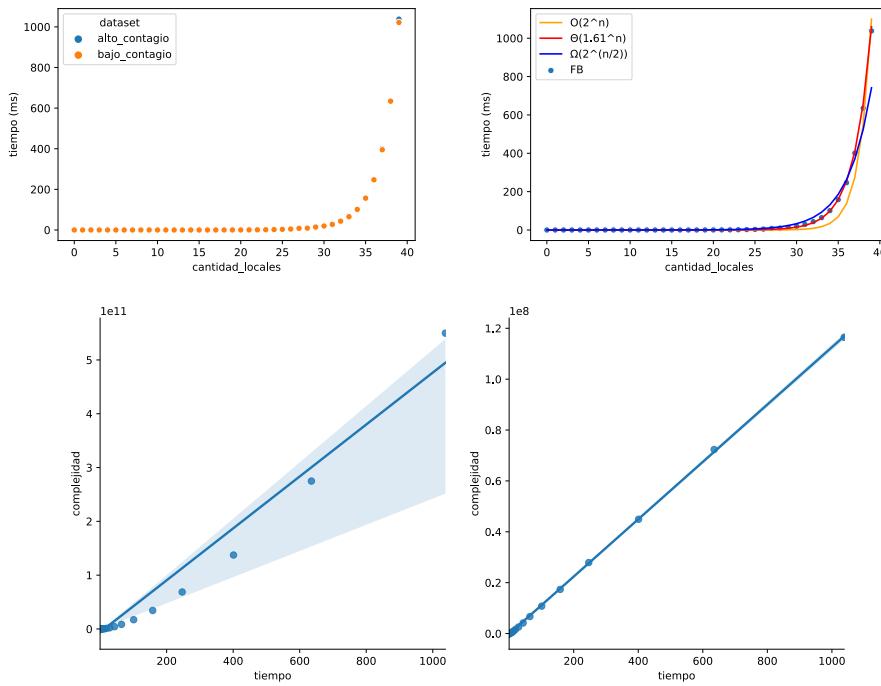


Figura 3

Figura 3 : Análisis de complejidad del método FB. El gráfico superior derecho corresponde a los tiempos de ejecución sobre los datasets. Mientras que el gráfico superior izquierdo contrasta estos tiempos de ejecución vs funciones de complejidad. Los inferiores son los gráficos de correlación entre los tiempos de ejecución contra la complejidad en el peor caso (derecha), y contra la complejidad obtenida (izquierda).

En el gráfico de arriba a la izquierda perteneciente a la Figura 3 se pueden observar los resultados del experimento, para los datasets utilizados donde claramente se evidencia un comportamiento exponencial. Por otro lado en el gráfico superior derecho tenemos el análisis de estos resultados en contraste con la curva para la constante que más se ajusta a nuestra complejidad 1.61^n y las curvas $\mathcal{O}(2^n)$ y $\mathcal{O}(2^{\frac{n}{2}})$. Se puede apreciar la exactitud con la que se solapan los tiempos de ejecución con la curva para la constante propuesta. Mientras que para las curvas $\mathcal{O}(2^n)$ y $\mathcal{O}(2^{\frac{n}{2}})$ nos encontramos que donde comienza su crecimiento exponencial se despegan, una por debajo y la otra superiormente. Cómo se puede apreciar se pudo obtener la constante que más adecúa a la

complejidad de nuestro algoritmo cumpliendo con las condiciones estipuladas en la hipótesis. Además como los resultados son los obtenidos por dos familias de *datasets* distintos, lo cual aporta una mayor certeza que en lugar de utilizar un mismo conjunto de instancias.

A continuación, tomamos la ejecución sobre el *dataset alto-contagio* y evaluamos cuál es su correlación con la complejidad en peor caso estudiada en la Sección 2.1 y la obtenida. En el gráfico inferior izquierdo de la Figura 3 se enumeran las instancias I_1, \dots, I_k y para cada una se grafica el tiempo de ejecución real $T(I_i)$ contra el tiempo esperado en peor caso $E(I_i) = 2^i$, es decir, su *gráfico de correlación*. De manera análoga en el gráfico inferior derecho lo hacemos para el tiempo esperado $E(I_i) = 1,61^i$ mostrando como se ajusta más el gráfico de correlación de esta en comparación de la anterior.

Se puede ver que el tiempo de ejecución sigue claramente una curva exponencial y además la correlación con la función 2^n es positiva y casi óptima. Sin embargo, la correlación con la función $1,61^n$ es significativamente mayor en comparación. En particular, el índice de correlación de Pearson entre el tiempo de ejecución y el peor caso es $r_{2^n} \approx 0,98457$, en contraste con el obtenido para la constante encontrada el cual es $r_{1,61^n} \approx 0,99991$. Por lo tanto, podemos afirmar que el algoritmo se comporta como se describió inicialmente en las hipótesis.

3.4. Experimento 2: Complejidad de Backtracking

Hipótesis 2. La complejidad en peor caso de BT es $\mathcal{O}(n2^n)$, para el caso intermedio que aprovecha solamente la poda de optimalidad es $\mathcal{O}(n^2)$ y finalmente para el mejor caso es $\mathcal{O}(n)$.

En esta experimentación vamos a contrastar las complejidades propuestas en la Sección 2.2 con respecto a las familias de instancias de mejor, caso intermedio y peor caso para el Algoritmo 2, y sus respectivas complejidades. Para esto evaluamos el método BT con respecto los *datasets* **bt-mejor-caso**, **bt-caso-intermedio** y **bt-peor-caso**.

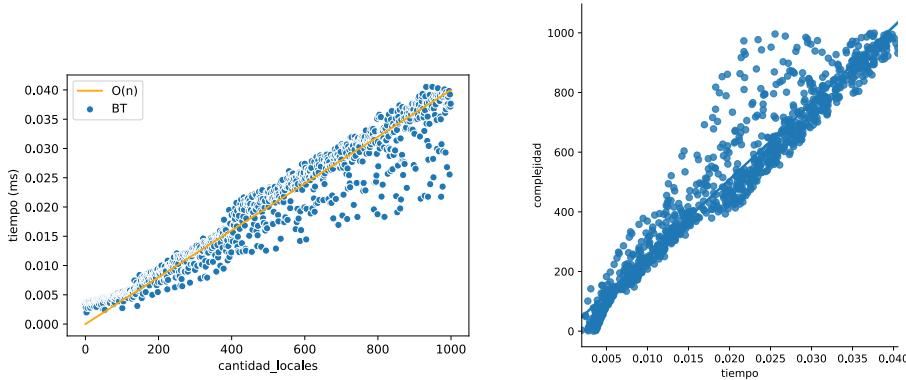


Figura 4: Análisis de complejidad del método BT para el data set bt-mejor-caso. A la derecha tenemos el gráfico de correlación entre el tiempo de ejecución y la complejidad esperada. Y a la izquierda Tiempo de ejecución vs Complejidad esperada.

Las Figuras 4, 5 y 6 muestran los gráficos de tiempo de ejecución de BT y de correlación para cada dataset respectivamente. Efectivamente, las hipótesis presentadas anteriormente se cumplen para los 3 casos. Por un lado, para las instancias de mejor caso se puede ver que efectivamente la serie de puntos muestra un crecimiento lineal aunque presenta cierto ruido". Uno de los motivos para este comportamiento es que al ser un comportamiento lineal, los tiempos de ejecución son muy bajos para incluso $n = 1000$. Como resultado, cualquier interferencia en el sistema operativo o cambio de contexto puede causar una fluctuación indeseada y alterar los resultados. Sin embargo, el índice de correlación de Pearson es $r \approx 0,96098$ lo cual muestra que hay una correlación positiva fuerte entre los tiempos de ejecución y una función lineal.

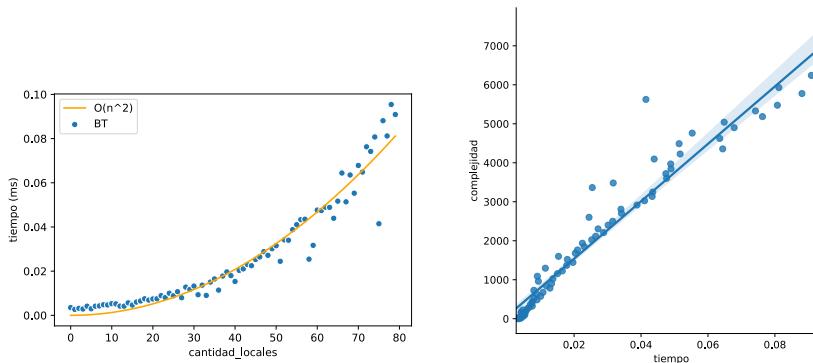


Figura 5: Análisis de complejidad del método BT para el data set bt-caso-intermedio. A la derecha tenemos el gráfico de correlación entre el tiempo de ejecución y la complejidad esperada. Y a la izquierda Tiempo de ejecución vs Complejidad esperada.

Mientras que en la Figura 5 para el caso intermedio nos encontramos también con “ruido” proveniente del sistema, pero igualmente con un comportamiento que se ajusta bastante a lo esperado. Ajustándose en gran proporción a la función cuadrática. A pesar del ruido podemos ver que nuestra complejidad es bastante acertada dado que el índice de correlación de Pearson obtenido es $r \approx 0,96910$ lo cual evidencia una correlación positiva y fuerte entre los tiempos de ejecución para este dataset y una función cuadrática.

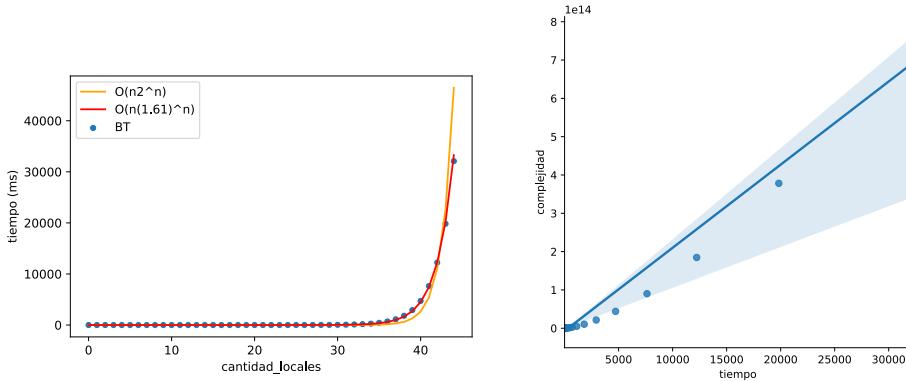


Figura 6: Análisis de complejidad del método BT para el data set bt-caso-intermedio. A la derecha superior tenemos el gráfico de correlación entre el tiempo de ejecución y la complejidad esperada. A la izquierda superior Tiempo de ejecución vs Complejidad esperada.

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 supra-exponencial. Notar que de manera anecdótica se muestra el ajuste a la complejidad que se obtiene utilizando la constante encontrada en los resultados del Experimento 1. Notemos que en este caso se ejecutaron instancias hasta $n = 45$, número elegido para evitar que el tiempo de ejecución sea demasiado grande (> 30 segundos). Para estas instancias el índice de correlación de Pearson es de $r \approx 0,98261$ contra la función $n2^n$. Por otro lado agregamos el Gráfico 10, en el apéndice, que muestra la correlación con la complejidad que se obtiene utilizando la constante obtenida en el primer experimento. Donde se puede ver un ajuste mucho más preciso con $r \approx 0,99981$ como el índice de correlación de Pearson obtenido.

3.5. Experimento 3: Efectividad de las podas

Hipótesis 3. El comportamiento del algoritmo va a variar según la densidad de contagio. Para instancias de alta densidad de contagio la efectividad de la poda de factibilidad se va a ver incrementada, mientras que la poda de optimalidad será menos efectiva. Por otro lado, para instancias de baja densidad de contagio la efectividad de la poda de factibilidad va a ser significativamente menor.

Es razonable asumir que a partir del experimento anterior surjan diversas preguntas. En particular, una podría ser de que manera se puede aprovechar cada poda en distintos escenarios, y qué factores impactan en el algoritmo cuando transita estos escenarios. Una de las hipótesis es que el Algoritmo 2 mejora su funcionamiento dependiendo de la densidad de contagio en las instancias, es decir, de cuántos locales se necesitan para igualar o superar el límite de contagio M . Por ejemplo, si $M = 100$ y $c_i > 50$ para todo i , entonces al seleccionar dos elementos en una solución parcial del algoritmo, o bien se iguala M o bien se supera ese valor y la poda por factibilidad es ejecutada. Por otra parte en instancias de baja densidad de contagio, si ocurre que para $n = M = 100$, $1 \leq c_i \leq 2$ y $b_i > 2$, esto nos dice que no tenemos restricción en la cantidad de locales a agregar, es decir, la poda por factibilidad se encuentra sin efectividad.

En este experimento vamos a comparar el funcionamiento de los métodos BT, BT-F y BT-O con respecto a los datasets de **alto-contagio** y **bajo-contagio**. En los gráficos superiores de la Figura 7 se muestran los resultados para el dataset **alto-contagio**. Se ejecutaron 10 instancias por cada $n \in 1, \dots, 150$ para evitar que el algoritmo demore más de unos segundos. Una observación interesante es que los tiempos de ejecución obtenidos para cada método se encuentran entre el peor y el mejor caso, vistos en el segundo experimento. Es interesante, sin embargo, observar los gráficos superior izquierdo e inferior izquierdo de la Figura 7 que hacen un acercamiento para evaluar mejor la diferencia entre BT y BT-O, y entre BT y BT-F. Podemos confirmar nuestra hipótesis viendo que en instancias de alto contagio el algoritmo BT-F supera al resto en tiempo de ejecución, siendo que las podas de factibilidad tienen un mayor impacto.

Por el lado de las instancias de baja densidad de contagio, los resultados están expuestos en los gráficos inferiores de la Figura 7. Lo más significativo de este gráfico es que a diferencia de las instancias con alta densidad de contagio, la poda de optimalidad gana en algunas instancias (con n entre 20 y 25) por sobre la de factibilidad, sin embargo podemos observar al acercarnos que sobre este tipo de instancias la combinación de ambas podas resulta ser lo más eficaz. En particular, esto se observa en el tamaño de las instancias ejecutadas (hasta $n = 45$) que si bien presentan tiempos más chicos que en las instancias de peor caso conservan su naturaleza exponencial. Como conclusión podemos afirmar que la densidad de las instancias tiene un peso significativo en el funcionamiento de este algoritmo.

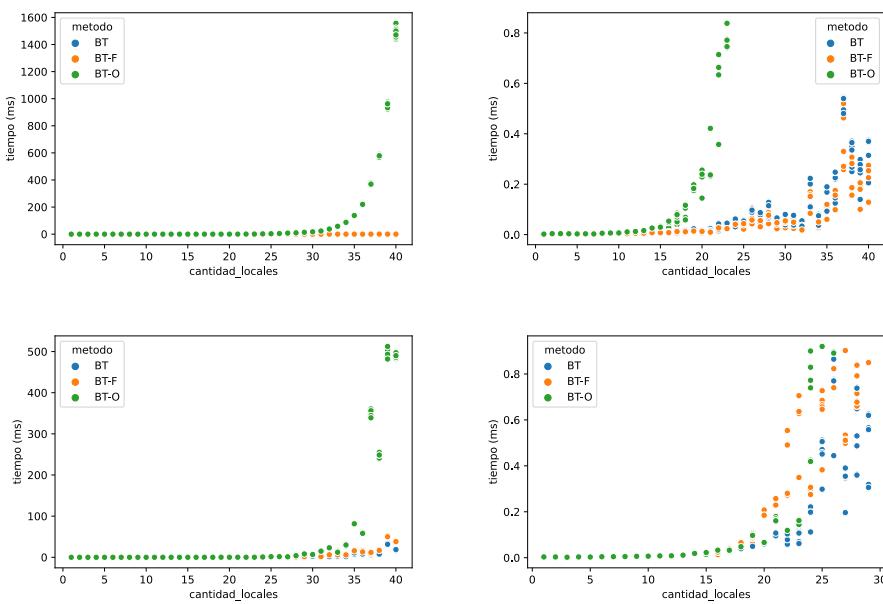
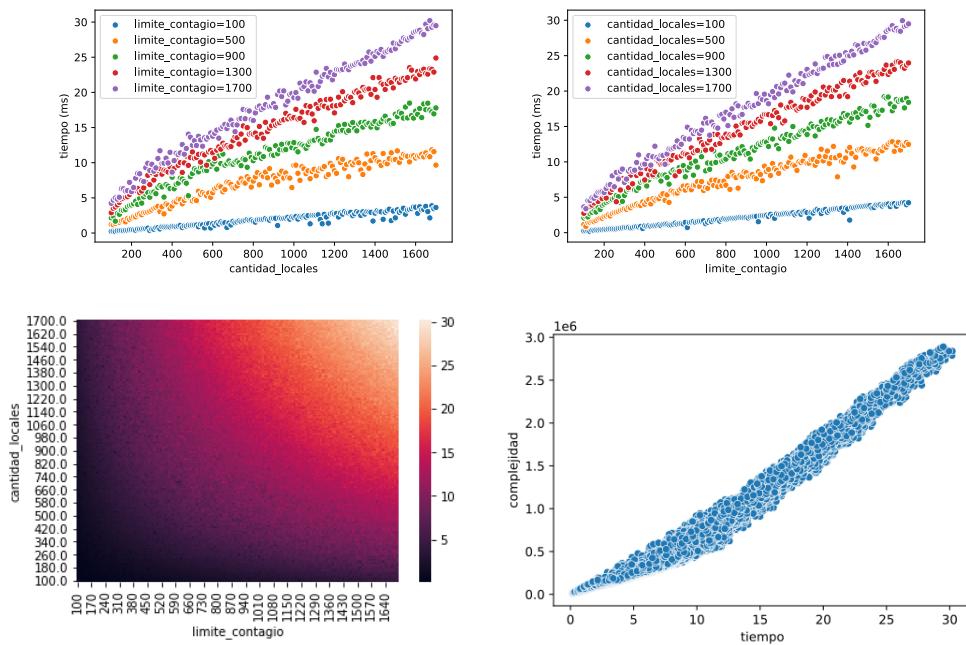


Figura 7: Comparación de efectividad en las podas.

3.6. Experimento 4: Complejidad de Programación Dinámica

Hipótesis 4. La complejidad del algoritmo de Programación dinámica va a ser $\mathcal{O}(n^*M)$

En este experimento vamos a analizar el comportamiento de nuestro algoritmo de Programación Dinámica. En la sección de Desarrollo, justificamos la complejidad de nuestro algoritmo de DP 3 llegando a la conclusión de que es $\Theta(n^*M)$. Para analizar esto en la práctica, lo ejecutamos con el dataset de **variación-n-m**.


 Figura 8: Resultados computacionales para el método DP sobre el dataset **variacion-n-m**.

Las primeras dos imágenes de la Figura 8 muestran el crecimiento del tiempo de ejecución en función de M y n respectivamente. Podemos observar que todas las líneas se comportan de forma similar, con un crecimiento lineal en función de ambas variables. Lo podemos reafirmar con el gráfico de heatmap, donde podemos apreciar que el crecimiento es similar tanto en dirección de n como de M. Para terminar, para confirmar que la complejidad algorítmica es la descripta anteriormente, tenemos en nuestro último

gráfico la correlación a lo largo de todas las instancias del tiempo de ejecución versus el tiempo esperado, que, como podemos observar, es bastante positiva, teniendo un índice de correlación de Pearson de $r \approx 0,98597$.

3.7. Experimento 5: Backtracking vs Programación Dinámica

Hipótesis 5. Cuanto mayor es el límite de contagio más ineficiente va a resultar DP con respecto a BT. Sin embargo, para límites de contagio muy bajos DP va a presentar una mayor eficiencia.

En nuestro último experimento, vamos a comparar los comportamientos del algoritmo de Backtracking y de Programación Dinámica cuando se enfrentan a distintas situaciones; por un lado, cuando la densidad de contagio es muy alta y M es significativamente mayor al n , y por el otro, cuando la densidad de contagio es baja con M menor a n . Vamos a analizar cuál es la mejor elección para cada situación. Para empezar, podemos observar que a mayor M , mayor será el costo de mantenimiento de las estructuras de memoización de nuestro algoritmo de Programación Dinámica, mientras que nuestro algoritmo de Backtracking podrá podar de forma más eficiente en instancias donde la densidad de contagio sea alta.

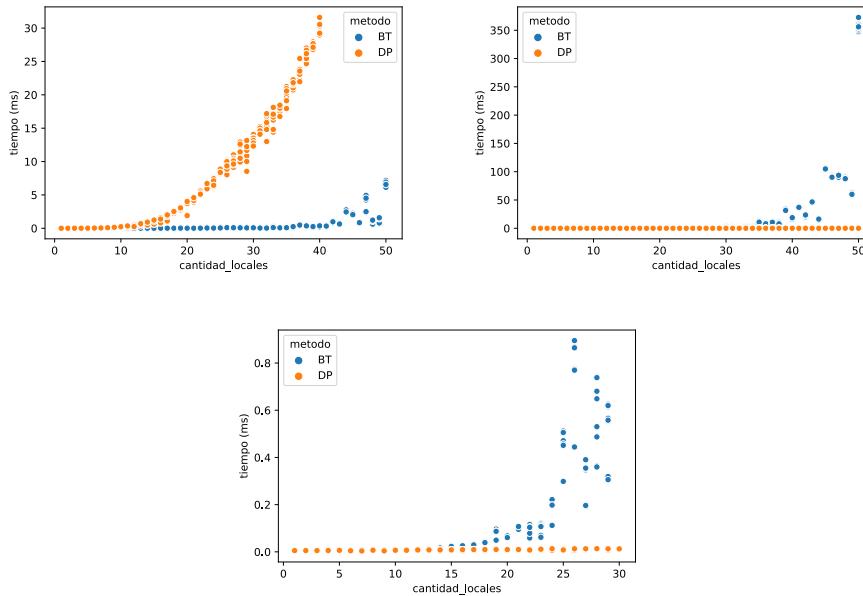


Figura 9: Comparación de tiempos de ejecución entre DP y BT. El gráfico superior izquierdo muestra los resultados para instancias con alta densidad. Mientras que el gráfico derecho lo hace para las instancias con baja densidad, mostrando un zoom de este en el gráfico inferior.

En la Figura 9 podemos ver que el crecimiento del algoritmo de Programación Dinámica para el dataset de alto contagio es exponencial, ya que el tiempo de ejecución de peor caso depende de M , por lo que confirmamos nuestra hipótesis de que en este tipo de instancias la elección más segura va a ser el algoritmo de Backtracking. Sin embargo, para instancias donde la densidad de contagio es baja, vemos que la estructura de memoización resulta mucho más eficaz que las podas y el tiempo de ejecución de DP se aplana respecto al de BT, por lo que la elección correcta sería el método de DP.

4. Conclusiones y trabajo futuro

En este trabajo se presentaron tres algoritmos que usan distintas estrategias para resolver NPM. El algoritmo de Fuerza Bruta, tanto para su mejor como peor caso tiene una complejidad de orden exponencial. Esto nos dice que resulta poco eficiente para resolver al problema, dado que a medida que aumenta el número de locales en L crece su tiempo de ejecución a valores enormes. Por otro lado, el algoritmo de Backtracking con la utilización de podas permite reducir significativamente los tiempos de ejecución para determinadas familias de instancias. Finalmente, el algoritmo de Programación dinámica es el que presenta una mayor robustez en términos de crecimiento de tiempos de ejecución para grandes valores de n , por otro lado las variaciones en los valores de M lo afectan significativamente, lo que determina que en escenarios donde el límite de contagio es muy alto puede no resultar tan eficiente.

Un posible trabajo a futuro, es explorar distintas podas de optimalidad para el algoritmo de Backtracking dado que ésta incrementa la complejidad temporal de manera significativa. A su vez, explorar nuevas podas que hagan más robusto al algoritmo en otros contextos que podrían hacer que se elija Backtracking por sobre nuestra implementación actual de DP. Por otro lado, para Programación Dinámica sería interesante realizar una implementación *bottom-up* y cambiar la estructura de memoización por una que reduzca el uso de memoria para instancias con un gran número de locales o límite de contagio. Por último, podríamos proponer

distintas métricas para un análisis más exacto de la complejidad en los experimentos, como por ejemplo conteo de instrucciones realizadas o la cantidad de nodos generados en el árbol recursivo. Esto podría aminorar el costo del análisis de datos, para poder lograr un resultado más concluyente con menos instancias.

5. Apéndice

5.1. Demostración de la complejidad de Fuerza Bruta

Veamos a que la complejidad del **Algorithm 1** en peor caso pertenece al orden $\mathcal{O}(2^n)$. Como se puede ver en los comentarios del algoritmo, la complejidad está dada por la cantidad de llamados recursivos, considerando que las operaciones de suma y comparación se realizan en $\mathcal{O}(1)$. Por lo tanto sea $T(n+1)$ la ecuación de recurrencia que describe la complejidad de nuestro algoritmo, para una cantidad de locales n :

$$\begin{aligned} T(0) &= \mathcal{O}(1) \\ T(1) &= \mathcal{O}(1) \\ T(n+1) &= T(n) + T(n-1) + \mathcal{O}(1) \end{aligned}$$

Podemos acotar superiormente $T(n+1)$ [5.1], sabiendo que $T(n-1) \leq T(n)$. Y sea la constante c , la cual acota el número de operaciones elementales que se realizan. Podemos establecer lo siguiente :

$$\begin{aligned} T(n+1) &\leq 2T(n) + c = \\ &= 4T(n-1) + 3c = \dots \\ &\dots = 2^{k+1}T(n-k) + (2^{k+1}-1)c \end{aligned}$$

Considerando que termina cuando $n-k+1 = 0 \implies k = n+1$. Entonces resulta que :

$$\begin{aligned} &= 2^{n+1}T(0) + (2^{n+1}-1)c \\ \implies T(n+1) &\leq (1+c)2^{n+1} - c \end{aligned}$$

Gracias a lo mostrado anteriormente se puede ver que $T(n+1) \in \mathcal{O}(2^n)$. Por otro lado por como está dada la recursión, en el mejor caso la complejidad es potencialmente menor al peor caso para ver esto. Veamos que podemos acotar inferiormente a $T(n+1)$, utilizando la misma desigualdad pero del otro lado :

$$\begin{aligned} T(n+1) &\geq 2T(n-1) + c = \\ 2(2T(n-3) + c) &= 4T(n-3) + 3c = \\ &= 8T(n-5) + 7c = \dots \\ &\dots = 2^kT(n-2k+1) + (2^k-1)c \end{aligned}$$

Considerando que termina cuando $n-2k+1 = 0 \implies k = \frac{n+1}{2}$. Entonces resulta que :

$$\begin{aligned} &= 2^{\frac{n+1}{2}}T(0) + (2^{\frac{n+1}{2}}-1)c \\ \implies T(n+1) &\geq (1+c)2^{\frac{n+1}{2}} - c \end{aligned}$$

Esto nos demuestra que $T(n+1) \in \Omega(2^{\frac{n}{2}})$

5.2. Influencia de los resultados en el Experimento 1 con el Experimento 2

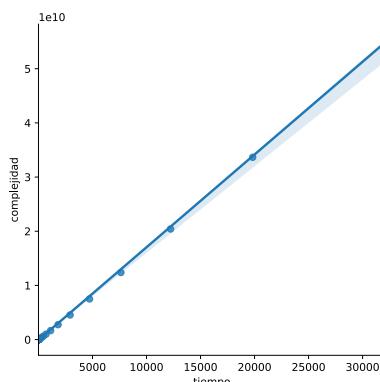


Figura 10: En este gráfico mostramos el Gráfico de correlación usando la constante encontrada en los resultados del Experimento 1