DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1

Distanciamiento Social

12 de septiembre de 2021 — Algoritmos y Estructuras de Datos III

Grupo 19

Integrante	LU	Correo electrónico
Damburiarena, Gabriel	889/19	gabriel.damburiarena@gmail.com
Guastella, Mariano	888/19	marianoguastella@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
http://www.exactas.uba.ar

1. Introducción

Uno de los mayores inconvenientes que hoy en día azotan a la humanidad es el, relativamente nuevo, virus SARS-CoV-2. Como respuesta inmediata a la rápidamente creciente pandemia, incluso las potencias mundiales tuvieron que establecer una cuarentena estricta poniendo, así, a la mayoría de comercios fuera de comisión. Aquellas entidades que no fueron determinadas como esenciales no tuvieron más remedio que temporalmente cesar actividades, muchas de las cuales quedaron al borde de la quiebra, si no lo hicieron completamente.
Luego de la primer ola de contagios, los gobiernos comenzaron a plantearse la reapertura de la actividad económica para evitar una crisis aún mayor. De ésta manera se creó un sistema para decidir qué negocios tendrían el privilegio de abrir, al cual llamaron "Negocio por Medio"(o NPM). Dentro de éste sistema, a cada negocio se le asignaría una cantidad de beneficio que aporta a la economia, y también cuánto contagio éste produciría. El objetivo de NPM sería encontrar una combinación de negocios a abrir que maximice el beneficio aportado y que no supere una cantidad de contagio máxima dada. Además, como lo indica su nombre, NPM no puede permitir que dos negocios contiguos abran. Formalmente, dada una secuencia de n negocios en orden $L = [1, \ldots, n]$, el beneficio y contagio b_i, $c_i \in N \geq 0$ de cada local $i \in L$, y el límite de contagio $M \in N \geq 0$, buscamos un subconjunto $L' \subseteq L$ tal que:

1. se maximice $\sum_{i \in L'} b_i$;
2. $\sum_{i \in L'} c_i \leq$ M;
3. no existe $i \in L'$ tal que $i + 1 \in L'$.

Por ejemplo:

Si $L = [1, 2, 3, 4]$, $C = [10, 20, 30, 40]$, $B = [50, 40, 10, 20]$, $M = 50$, la solución óptima sería $L' = [1, 4]$ con un beneficio total de 70.

Si $L = [1, 2, 3, 4, 5]$, $C = [20, 10, 50, 30, 40]$, $B = [20, 35, 10, 20, 70]$, $M = 100$, la solucion óptima sería $L' = [2, 5]$ con un beneficio total de 105.

El objetivo de este trabajo práctico es utilizar tres técnicas algorítmicas diferentes para resolver problemas y evaluar su eficacia en diversas instancias de problemas. En primer lugar, utilizamos la Fuerza Bruta, que implica enumerar todas las soluciones posibles y encontrar de forma recursiva soluciones factibles. Luego, se introduce la poda para reducir el número de nodos en el árbol recursivo para encontrar un algoritmo más efectivo, a fin de obtener un algoritmo de Backtracking. Finalmente, la introducción de tecnología de memoria para evitar cálculos repetidos de subproblemas. La última técnica se llama Programación Dinámica.
El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2 se define el algoritmo recursivo de Fuerza Bruta para recorrer todo el conjunto de soluciones y se analiza su complejidad. Más tarde, en la Sección 3 se explica el algoritmo de Backtracking con un breve análisis de mejores y peores casos. Luego, se introduce el algoritmo de Programción Dinámica en la Sección 4 junto con la demostración correspondiente de correctitud y un análisis de complejidad. Finalmente, en la Sección 5 se presentan los experimentos computacionales, y las conclusiones finales se encuentran en la Sección 6.

2. Fuerza Bruta

El algoritmo de Fuerza Bruta enumera todo el conjunto de soluciones para encontrar una solución factible u óptima según si el problema es un problema de decisión o un problema de optimización. En este caso, el conjunto de soluciones está compuesto por los posibles beneficios que se pueden obtener de L. Por ejemplo, si $L = [1, 2, 3], C = [50, 40, 30], B = [40, 30, 30]$, $M = 70$, el conjunto de soluciones factibles es $L'' = [[1], [2], [3]]$ y la solución óptima es $L' = [1]$ cuyo beneficio es $b_1 = 40$.
La idea del FuerzaBruta para resolver NPM es ir generando soluciones de manera recursiva decidiendo en cada paso si toma o no un negocio de L, agregándolo a la solución parcial negocios y quedandose con el máximo entre el beneficio obtenido en ambos caminos. Por simplicidad vamos a combinar los beneficios y los contagios en una tupla de negocios dentro de L. Al llegar a una hoja del árbol de recursión, es decir un caso base, determina si es una solución factible chequeando que no se pase del valor de contagio máximo y que no sean contiguos, de ser así devuelve el beneficio obtenido, en caso contrario devuelve 0.

En el siguiente algoritmo (y en aquellos de las otras técnicas algoritmicas), se utilizará un vector de pares como estructura de respresentación de las secuencias L, C y B, al cual llamaremos L. Cada par (x, y) representaría un negocio con beneficio 'x' y contagio 'y'. También notar que L y M se van a pasar por referencia.

FuerzaBruta(L, M, i, negocios)

```
1: if i = |L| then
2:     for j =0 hasta |negocios| − 1 do
3:         if negocios[j] + 1 = negocios[j + 1] then
4:             return 0
5:         end if
6:     end for
7:     b_acumulado = 0
8:     m_acumulado = 0
9:     for negocio in negocios do
10:        b_acumulado = b + primero(L[negocio])
11:        m_acumulado = c + segundo(L[negocio])
12:    end for
13:    if m_acumulado > M then
14:        return 0
15:    end if
16:    return b_acumulado
17: end if
18: return max(FuerzaBruta(L, M, i + 1, negocios), FuerzaBruta(L, M, i + 1, negocios ∪ i))
```

En la Figura 1 se ve el ejemplo antes mencionado. Cada nodo intermedio del árbol representa una solución parcial, es decir, cuando todavía no se decidió qué elementos incluir, mientras que las hojas representan a todas las soluciones.

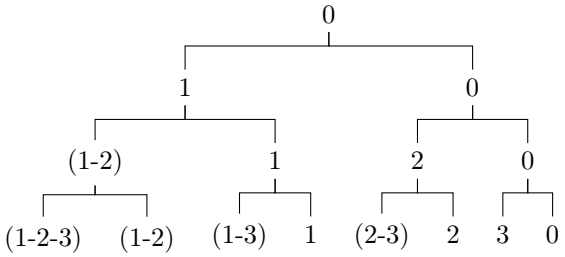

Figura 1: Ejemplo de ejecución de FuerzaBruta para $L = [1, 2, 3], C = [50, 40, 30], B = [40, 30, 30],\ M = 70$, notar que las soluciones 1, 2, 5 no cumplen con la condición de no ser contiguas, la 3 con el máximo de contagios y el resto son soluciones factibles. La solución óptima es la 4ta.

La correctitud del algoritmo se basa en que se generan todas las posibles soluciones, dado que para cada negocio en L se crean dos ramas: una en la que se lo incluye y otra en la que no. Al haberse generado todas las soluciones, se va a devolver la óptima.
La complejidad de FuerzaBruta para el peor caso es $\mathcal{O}(n * 2^n)$. Esto ocurre porque el árbol de recursión es un árbol binario completo de $n + 1$ niveles (contando la raíz), dado que cada nodo se divide en dos hijos y en cada paso el parametro i es incrementado en 1 hasta llegar a n, esto es $\mathcal{O}(2^n)$. Luego, cuando se llega a una hoja, recorre todo la solución parcial que se tiene para evalúar si es factible, esto es $\mathcal{O}(n)$. Finalmente, la complejidad es $\mathcal{O}(n * 2^n)$.

3. Backtracking

Se podría decir que un algoritmo de Backtracking es una 'mejora' de un algoritmo de fuerza bruta. Es decir, en el Backtracking se forma un arbol similar al de Fuerza Bruta, en el cual cada nodo genera tantos nodos como decisiones locales haya para tomar y luego se obtiene la solución óptima hayada en alguna de las ramas. Dónde los algoritmos difieren es cuando, en el caso del Backtracking, se implementan las 'podas', que sirven como reglas para evitar explorar partes del árbol en las que se sabe que no se encotrará una solución óptima. Las podas, generalmente, suelen dividirse en dos categorías: *factibilidad* y *optimalidad*.

Poda por factibilidad. En éste caso la poda por factibilidad se efectúa de ésta manera: dada una solución parcial L' tal que $\sum_{i \in L'} c_i = m$ y una cantidad máxima de contagio admitida M, si en algun nodo se registra que $m > M$, entonces no habría manera de extender L' tal que $m \leq M$, es decir, cualquier solución que parta del nodo que tiene

como solución parcial a L' no será factible, ya que L' no lo es. De ésta manera evitamos explorar una parte del árbol y así reducir la cantidad de operaciones que el algoritmo ejecuta.

Poda por Optimalidad. La poda por optimalidad entra en juego cuando se obtiene al menos una solución válida. En nuestro problema en particular, dada una solucion válida V y una solución parcial L' representada por un nodo intermedio n_0, decimos que $B = \sum_{j \in V} b_j$ es el beneficio total de la solución válida, $b = \sum_{j \in L'} b_j$ el beneficio total de la solución parcial, y llamaremos i a la iésima iteracion del algoritmo, que corresponde al nodo n_0. Si $b + \sum_{j=i}^{n} b_i \leq B$, es decir, si el beneficio total de la solución parcial más la sumatoria del beneficio de todos los negocios entre i y n no supera el beneficio de la solución ya encontrada, entonces se puede afirmar que no hay combinación de negocios en esa rama, que nos interese. De ésta manera podemos podar el nodo n_0 y ahorrar tiempo de ejecución.

```
1: B = 0
2: Backtracking(L, M, b_restante, i, b, m)
3: if m > M then
4:     return 0
5: end if
6: if b + b_restante ≤ B then
7:     return b
8: end if
9: if i ≥ |L| then
10:    if m > M then
11:        return 0
12:    end if
13:    B = max(B, b)
14:    return b
15: end if
16: return max(Backtracking(L, M, b_restante − primero(L[i]), i + 1, b, m),
17:            Backtracking(L, M, b_restante − primero(L[i]), i + 2, b + primero(L[i]), m + segundo(L[i])))
```

Algunas cosas para notar: 'b_restante' es el beneficio acumulado de todos los negocios de L y se calcula $\mathcal{O}(n)$ previo al algoritmo. 'b' es el beneficio acumulado de la solución parcial, 'm' es el contagio acumulado de la solución parcial.
La complejidad de este algoritmo en peor caso es de $\mathcal{O}(2^n)$ ya que si no es posible hacer alguna poda, entonces habrá tantas llamadas recursivas ejecutadas como nodos en el arbol de recursión completo de Fuerza Bruta. Para que esto suceda, La sumatoria de todos los contagios de los negocios deberia ser menor a M y los negocios deberían estar ordenados de mayor beneficio a menor beneficio. La complejidad de cada llamada recursiva individualmente sería $\mathcal{O}(1)$ ya que no se ejecuta ningun ciclo en la función y los parámetros L y M se pasan por referencia. En el mejor caso, el primer negocio de la secuencia L tendría beneficio $b_1 \geq n$ y contagio $m_1 = M$, y los siguientes n − 1 negocios tendrían $m_i > 0$ y $\sum_{i=2}^{n} b_i < b_1$. De esta manera, el algoritmo se podrá dividir en dos casos: el caso en el que se incluye el primer negocio a la solución (1) y el caso en el que no se incluye (2).
(1). Si se incluye el primer negocio a la solución, no se puede incluir ningún otro, es decir, se podan todas las ramas que agreguen un nuevo negocio. Ésto costaría una llamada recursiva por cada negocio, es decir, sería $\mathcal{O}(n)$. Al ejecutarse por completo este caso, se obtendría la solución óptima b_1.
(2). Si no se incluye el primer negocio, tendríamos que b_restante ahora es menor a la solución b_1 ya encontrada, por lo que se devuelve 0 y la ejecución de éste caso termina en $\mathcal{O}(1)$.
La complejidad total para el mejor caso es, entonces, $\mathcal{O}(n)$.

4. Programación Dinámica

Los algoritmos de Programación Dinámica entran en juego cuando un problema recursivo tiene superposición de subproblemas. La idea consiste en evitar recalcular todo el subárbol correspondiente si este ya fue calculado con anterioridad. En este caso, definimos la siguiente funcion recursiva que resuelve el problema:

$$f(i, m, b) = \begin{cases} 0 & m > M \\ b & i \geq n \\ \max(\text{f}(i+1,\ m,\ b),\ \text{f}(i+2,\ m+c_i,\ b+b_i)) & \text{en caso contrario} \end{cases}$$

Correctitud:
(i) Si $m > M$ entonces el subconjuntos de negocios que elejimos no es valido y el beneficio es nulo, entonces la respuesta de $f(i, m) = 0$.
(ii) Si $i \geq n$ entonces quiere decir que llegamos al final del conjunto de negocios, por lo cual no hay ningún negocio para tener en cuenta. Por lo tanto, la respuesta es $f(i, m) = b$.
(iii) En este caso, $i < n$ y $m \leq M$ entonces estamos buscando el máximo beneficio un subconjunto de L. De existir el subconjunto, tiene que o bien tener al i-ésimo elemento o no tenerlo. Si lo tiene, entonces tiene el b_i en su beneficio total, por ende, se lo sumamos a la solución. Notar que salteamos el negocio contiguo en la siguiente llamada recursiva. Si no lo tiene seguimos y probamos con el siguiente. Es importante destacar que queremos encontrar el máximo beneficio, por lo cual nos vamos a quedar con el máximo entre ambos caminos.

Memoización: Vemos que la función recursiva toma dos parámetros $i \in [0, ...,n-1]$ y $m \in [0, ..., M]$. Notar que los casos $i = n + 1$ o $m > M$ son casos base y se resuelven en tiempo constante. Entonces, la cantidad de posibles insatancias con al que se puede llamar a la función está determinada por la combinación de ellos. Luego tenemos $\Theta(n * M)$ combinaciones posibles de parámetros. Luego, si agregamos una memoria que recuerde los casos ya resueltos y su correspondiente resultado, podemos calcularlos una sola vez y asegurarnos no resolver más de $\Theta(n * M)$ casos. El algoritmo ProgramaciónDinámica aplica esta idea.

```
1: Beneficios = ⊥ for i ∈ [0, |L|], m ∈ [0, M + 1]
2: ProgramaciónDinámica(L, M, i, b, m)
3: if m > M then
4:     return 0
5: end if
6: if i ≥ |L| then
7:     return b
8: end if
9: if Beneficios[i][m] = ⊥ then
10:    Beneficios[i][m] = max(ProgramaciónDinámica(L, M, i + 1, b, m),
11:            ProgramaciónDinámica(L, M, i + 2, b + primero(L[i]), m + segundo(L[i])))
12: end if
13: return Beneficios[i][m]
```

La complejidad del algoritmo entonces está determinada por la cantidad de estados que se resuelven y el costo de resolver cada uno de ellos. A lo sumo se resuelven $\mathcal{O}(n * M)$ estados distintos, y como todas las líneas del algoritmo ProgramaciónDinámica realizan operaciones constantes entonces cada estado se resuelve en $\mathcal{O}(1)$. Por ende, el algoritmo tiene complejidad $\mathcal{O}(n*M)$ en el peor caso. Es importante observar que el diccionario Beneficios se puede implementar como una matriz con acceso y escritura constante. Es más, notar que su inicialización tiene costo $\Theta(n * M)$, por lo tanto, el mejor y peor caso de nuestro algoritmo va a tener costo $\Theta(n * M)$.

5. Experimentación

En esta sección se presentan los experimentos computacionales realizados para evaluar los distintos métodos presentados en las secciones anteriores. Los mismos fueron realizados en una workstaticon con CPU AMD fx-8350 @ 4.2Ghz, 16 RAM y utilizando el lenguaje de programación python, el cual ejecuta los algoritmos que están escritos en C++ con el input adecuado.

5.1. Métodos

Las configuraciones y métodos utilizados durante fase de experimentación son los siguientes:

- **FB**: Algoritmo 1 de Fuerza Bruta de la sección 2.
- **BT**: Algoritmo 2 de Backtracking de la sección 3.
- **BT-F**: Algoritmo 2 pero únicamente utilizando la poda de factibilidad.
- **BT-O**: Algoritmo 2 pero únicamente utilizando la poda de optimalidad.

- **DP**: Algoritmo 3 de Programación Dinámica de la sección 4.

5.2. Instancias

Para poder evaluar la efectividad y el rendimiento de los algoritmos implementados es necesario someterlos a pruebas con distintas familias de instancias, cada cual con sus debidas características. En primer lugar vamos a definir la *densidad* de una instancia como el cociente $\frac{maxC_i}{M}$ de tal manera que si hay densidad baja, los números serán chicos en comparación con M, y si hay densidad alta, serán grandes en comparación con M.

- **densidad-alta**: En ésta instancia los negocios serán (1, 1), ... , (1, $i+1$) , ... , (1, $n+1$) en orden aleatorio con $M = \frac{n}{2}$.
- **densidad-baja**: En ésta instancia los negocios serán (1, 1), ... , (1, $i+1$) , ... , (1, $n+1$) en orden aleatorio con $M = \frac{n*(n-1)}{4}$.
- **bt-mejor-caso**: Para el mejor caso de bt se tiene $L = \{(n, M), (1, 1), ... , (1, 1)\}$
- **bt-peor-caso**: Para el peor caso de bt se tiene $L = \{(1, 1), ... , (i+1, 1), ... , (n+1, 1)\}$ con $M = n$.
- **dinamica**: Para dinámica, las instancias son todas con negocios (1, 1), pero varía el n y el M entre 1000 y 8000, salteando de a 500 entre instancias.

5.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento se analiza el rendimiento del metodo de Fuerza Bruta en distintas instancias. El análisis de la complejidad realizado previamente indica que el peor y mejor caso son idénticos y es exponencial en función de n, todo multiplicado por n. Para comprobar esto ejecutamos el algoritmo de Fuerza Bruta con los datasets densidad alta y baja y graficamos el tiempo de ejecución en función de n.
En la Figura 1 se puede apreciar que ambas curvas se solapan para la mayoría de casos. Lo que se muestra es que el tiempo de ejecución no depende de las características de las curvas.
En la Figura 3 evaluamos la correlación entre la complejidad estudiada en la Sección 2 y los resultados de las experimentaciones. Ambas son $\mathcal{O}(n * (2^n))$. En particular el coeficiente de Pearson da ≈ 0.9993 y se lo puede ver en la Figura 2. Por lo tanto, podemos afirmar que el algoritmo se comporta según lo esperado.

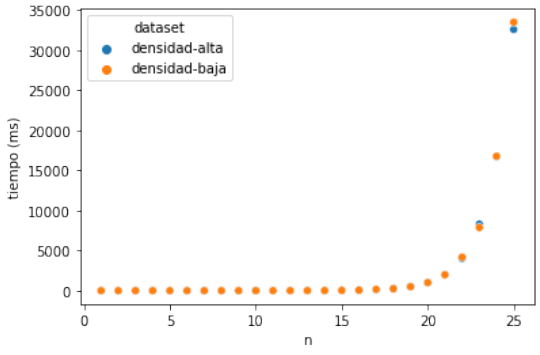

Figura 1: FB con ambas densidades

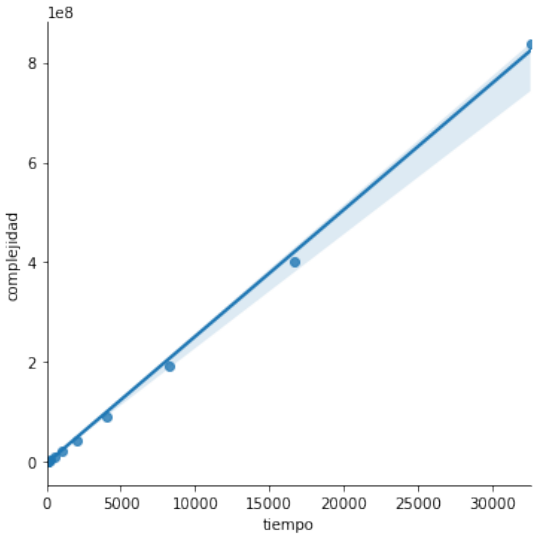

Figura 2: Correlación entre la experimentación y lo esperado

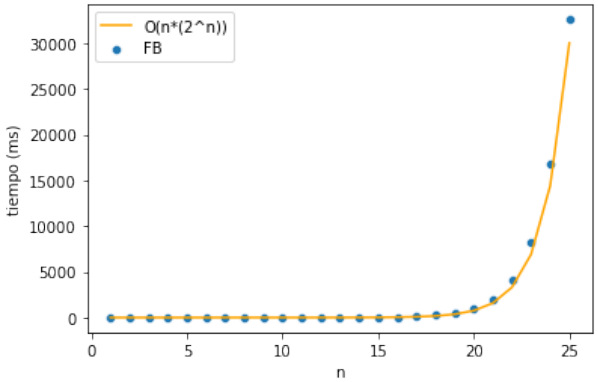

Figura 3: Tiempo de ejecución contra complejidad esperada

5.4. Experimento 2: Complejidad de Backtracking

En esta subsección vamos a contrastar las hipótesis hechas en la Sección 3 con las familias de instancias de mejor y peor caso de Backtracking y su esperada complejidad.
En el mejor caso (Figuras 4 y 5) se puede observar que los datos obtenidos a partir de las instancias son fieles a la complejidad lineal propuesta y las irregularidades se alejan muy poco de lo normal. El mejor caso de backtracking es único entre las otras familias de instancias ya que presenta tiempos extremadamente rápidos incluso para $n = 1000$. La correlación de Pearson para este caso es de 0.975 porque cuando se trabaja con tiempos más rápidos, cualquier fluctuación de uso del CPU, por parte del sistema operativo, puede causar fluctuaciones que separen al tiempo de ejecución de su valor esperado. Pero a grandes rasgos se puede concluir que los datos obtenidos siguen una complejidad lineal, es decir, la hipótesis propuesta es válida.
En nuestro otro caso, en cambio, estamos contemplando el peor caso (Figuras 6 y 7). Dado que no se realizan podas, la complejidad del algoritmo de Backtracking en esta familia de instancias resulta exponencial. Por esta misma razón es que no evaluamos $n > 30$, ya que el tiempo de ejecución resultaría demasiado grande. En la práctica, los datos parecen diferir en una muy pequeña medida de una exponencial con base 2, por lo que la correlación de Pearson es de ≈ 0.984. Esto se puede deber a alguna variación computacional inesperada, que termina reduciendo la base de la exponencial en la práctica. De todas maneras la complejidad observada es exponencial y podemos afirmar que la hipótesis es válida.

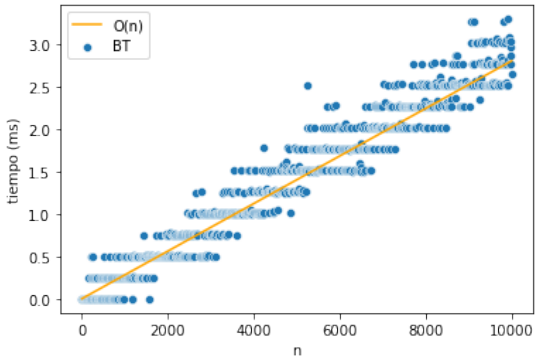

Figura 4: Tiempo de ejecución vs. complejidad esperada.

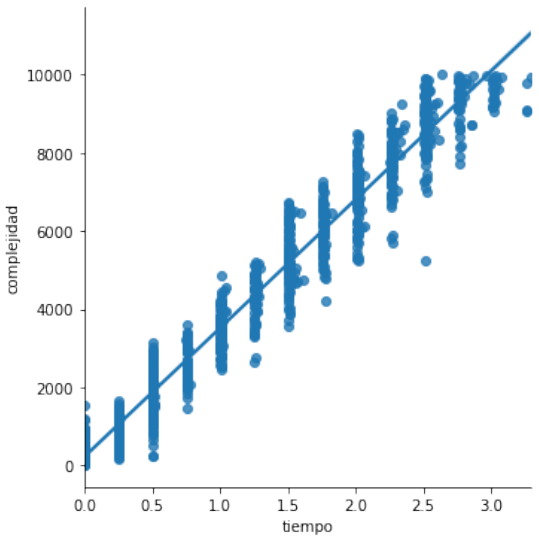

Figura 5: Correlación entre el tiempo de ejecución y la complejidad esperada.

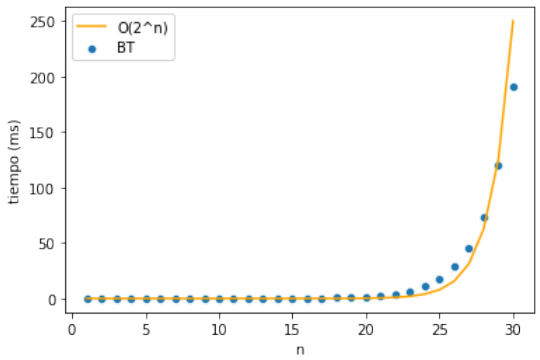

Figura 6: Tiempo de ejecución vs. complejidad esperada.

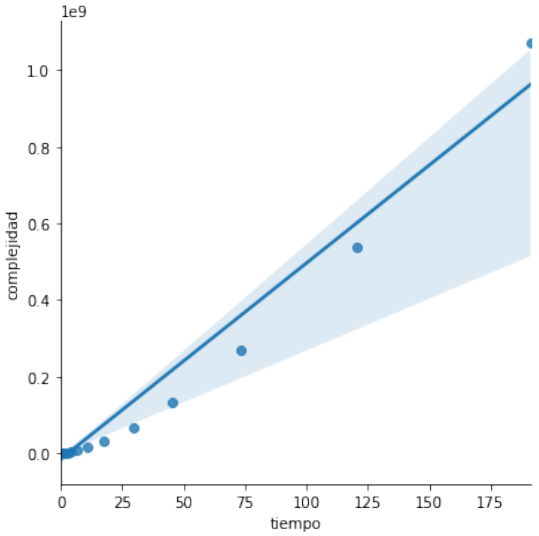

Figura 7: Correlación entre el tiempo de ejecución y la complejidad esperada.

5.5. Experimento 3: Efectividad de las podas

Luego de analizar las familias del mejor y peor caso, resta analizar los tiempos de ejecución de todas las instancias que quedan en el medio entre lo lineal y lo exponencial. Para ello, utilizaremos las familias de alta y baja densidad para evaluar la efectividad de los algoritmos 'BT', 'BT-F' y 'BT-O'. En primer lugar, la hipótesis respecto a las instancias de densidad alta vs. las de densidad baja, es que se ejecutarán más rápido las de densidad alta ya que la poda de factibilidad se aplicaría una mayor cantidad de veces. Esta hipótesis se puede confirmar al observar las figuras 8 y 10 (Se evito usar n >25 para que el tiempo de ejecución no sea muy alto). Con $n = 20$ (Excluyendo BT-O) en la figura 9 se tiene un tiempo que roza los 0ms, pero en la figura 11 se puede observar cómo el tiempo comienza a aumentar a partir de $n = 12$. Se excluyó BT-O de este analisis ya que la diferencia de tiempo se debe a la poda de factibilidad. En segundo lugar, es necesario comparar la efectividad de cada poda en alguna familia en particular. Observando la familia de densidad baja, se puede concluir que la poda de optimalidad tiene un impacto muy pequeño en la reducción de la complejidad. Esto puede deberse a que la densidad se definió respecto al contagio de los negocios. Por esta misma razón, los algoritmos BT y BT-F muestran tiempos similares, ya que la poda más significativa es la de factibilidad, que logra reducir la ejecución a valores insignificantes en comparación al de fuerza bruta. Esto se debe a que la poda de factibilidad se realiza en aproximadamente la mitad de las llamadas recursivas cuando es necesario saltear los negocios adyacentes y, en cambio, la poda de optimalidad es muy circunstancial ya que depende del orden de los negocios con respecto a su beneficio. Como la poda de optimalidad no agrega complejidad a la ejecución del algoritmo, y se probó que sirve para podar ramas muy complejas en tiempo constante en el análisis del mejor caso, se puede afirmar que el algoritmo es más efectivo cuando se utiliza esta poda.
Es importante notar que ambas instancias se encuentran entre el mejor y peor caso como era esperado.

5.6. Experimento 4: Complejidad de la Programación Dinámica

A continuación se analiza la eficiencia del algoritmo de Programación Dinámica en la práctica y su correlacion con la cota teórica calculada previamente. Para lograr esto, se corren las instancias del dataset dinamica sobre el método DP y se grafican sus resultados.
En las Figuras 12 y 13 se muestran el crecimiento del tiempo de ejecución de n y M respectivamente, sobre alguno cortes hechos en la otra variable. Como preevimos las complejidades en la práctica son las esperadas. Se puede ver que todas las líneas se comportan de una manera similar. Luego la complejidad es $\mathcal{O}(n * M)$. También se puede observar el mapa de calor en la Figura 14 y ver como varia el tiempo de ejecución en función de ambas variables al mismo tiempo. Claramente el crecimiento es similar para ambas variables. Por último, el coeficiente de Pearson es ≈ 0.9983 y en la Figura 15 se puede ver la correlación entre lo esperado teóricamente y lo obtenido en la experimentación. Es importante notar que es muy probable que se desperdicie memoria al utilizar este algoritmo ya que no siempre se van a calcular todas las posibles variaciones.

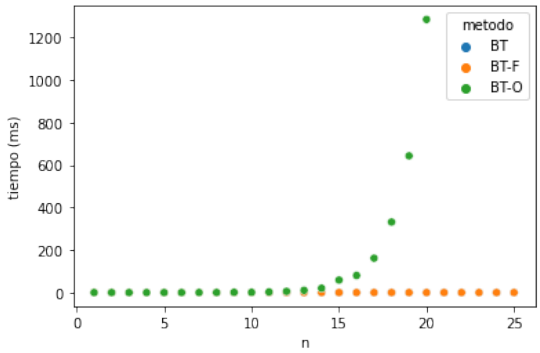

Figura 8: Comparación entre los distintos algoritmos de backtracking con densidad alta.

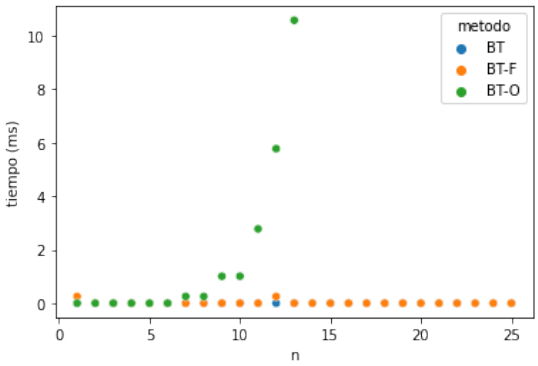

Figura 9: Zoom del gráfico de BT con densidad alta.

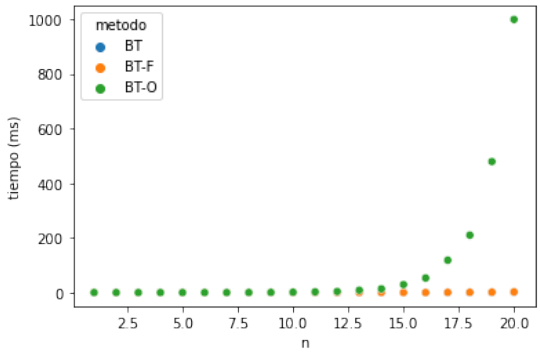

Figura 10: Comparación entre los distintos algoritmos de backtracking con densidad baja.

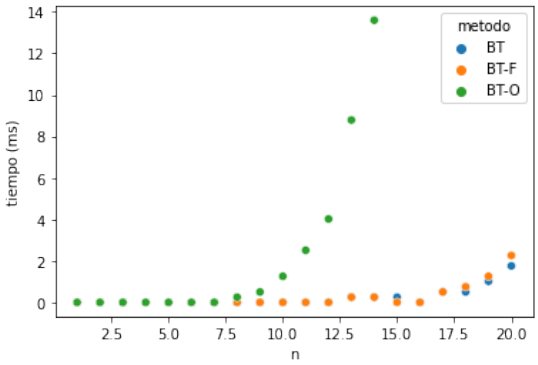

Figura 11: Zoom del gráfico de BT con densidad baja.

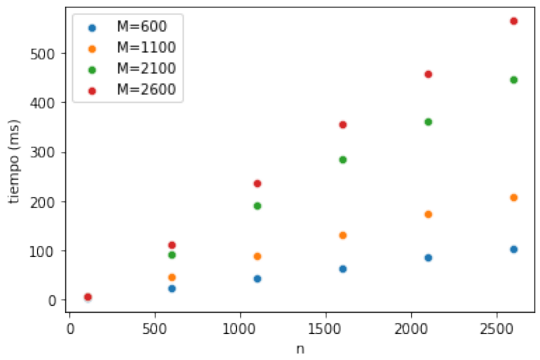

Figura 12: Crecimiento del tiempo de ejecución con cortes en M

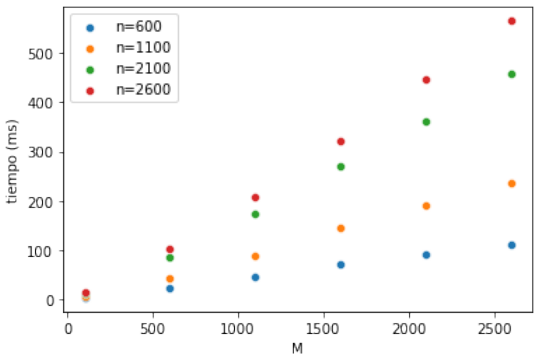

Figura 13: Crecimiento del tiempo de ejecución con cortes en n

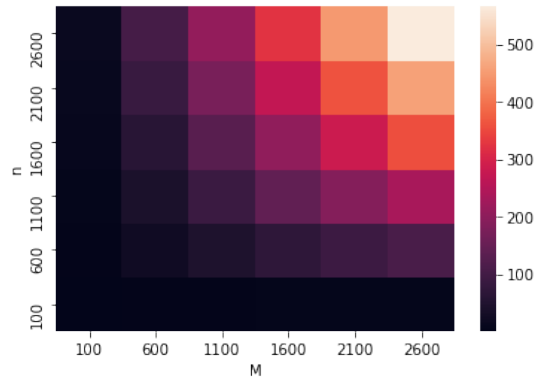

Figura 14: Mapa de calor

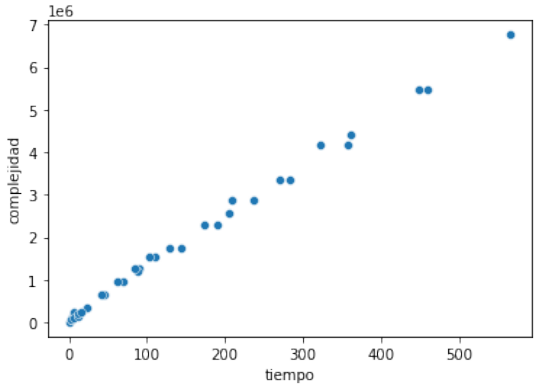

Figura 15: Correlación

6. Conclusiones

Terminada la sección de experimentación y probadas nuestras hipótesis, se puede comenzar a entablar una conclusión. En primer lugar pudimos ver que la fueza bruta es 'brutalmente' ineficiente para resolver este problema ya que al aumentar el tamaño de L el tiempo de ejecución se dispara a valores cada vez más grandes. La evolución natural de Fuerza Bruta es el Backtracking que, al introducir sus fabulosas podas, logra tiempos espectaculares, en comparación, y también muestra tiempos particulares para ciertas estructuras. Por último, tenemos el algoritmo de Programación Dinámica que logra calcular en segundos instancias con n mayor a cualquier otro algoritmo y por esta misma razón concluimos que es el mejor de todos los algoritmos descritos, aunque sea el que más se ve afectado por el crecimiento de M y a pesar de ser el que más memoria consume.
Cabe aclarar, en el caso del Backtracking, que es posible implementar podas más complejas que reduzcan la complejidad aún más, sobre todo en el caso de la poda de optimalidad.