DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Distanciamiento social

Algoritmos y Estructura de Datos III
Segundo Cuatrimestre de 2020

Grupo 29

Integrante	LU	Correo electrónico
Cerdeira, Elías Nahuel	692/12	eliascerdeira@gmail.com
Panichelli, Manuel	72/18	panicmanu@gmail.com

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

En tiempos de crisis sanitaria como los que atraviesa el país y el mundo en este momento, es necesario poder tomar decisiones que permitan resguardar la salud de la población y mantener el nivel de productividad económica que mitigue los efectos sobre los niveles de desocupación y pobreza. El contexto de *distanciaciamiento social* plantea el problema de *negocio por medio* (NPM) que busca determinar qué locales conviene abrir, dentro de un conjunto de locales de zonas comerciales, en función del *beneficio* económico que aportan, asignado por el gabinete económico, y un valor de *contagio*, asignado por un grupo de expertos de la salud.

Formalmente, dado una secuencia de $n \geq 0$ locales comerciales en orden $L = [1, \ldots, n]$, el beneficio y contagio $b_i, c_i \in \mathbb{N}_{\geq 0}$ de cada local $i \in L$ y el límite de contagio $M \in \mathbb{N}_{\geq 0}$, el problema de NPM consiste en determinar cuál es el máximo beneficio comercial obtenible sin sobrepasar el límite de contagio establecido por los especialistas. Para simplificar, se considera *solución* a todo subconjunto de locales $L' \subseteq L$ y se dice que es *factible* si la suma de los valores de contagio de los locales no supera el límite, es decir, (i) $\sum_{i \in L'} b_i \leq M$, y si no cuenta con dos locales colindantes, es decir, si (ii) $(\forall i \in [2, \ldots, n] \ \wedge \ i \in L') : (i - 1 \notin L')$. Además se espera que siempre sea posible al menos poder abrir un local, es decir, $\exists\, i \in [1, \ldots, n] : c_i \leq M$.

A continuación se exhiben algunos ejemplos con sus correspondientes respuestas esperadas.

- $n = 4,\ M = 40,\ b = [10, 20, 30, 40],\ c = [10, 10, 10, 10]$.

  Las soluciones factibles son $L'_1 = \{1\},\ L'_2 = \{2\},\ L'_3 = \{3\},\ L'_4 = \{4\},\ L'_5 = \{1, 3\},\ L'_6 = \{1, 4\},\ L'_7 = \{2, 4\}$ y la solución óptima es la 7 con beneficio máximo de 60.

- Por otro lado, si se tiene $M = 20,\ b = [10, 15, 30, 15],\ c = [15, 25, 10, 5]$.

  Las soluciones factibles son $L'_1 = \{1\},\ L'_2 = \{3\},\ L'_3 = \{4\},\ L'_4 = \{1, 4\}$ y la solución óptima es la 2 con beneficio máximo de 30.

El objetivo de este trabajo es implementar una solución para NPM utilizando tres técnicas algoritítmicas distintas y evaluar la efectividad de cada una para distintos conjuntos de instancias. En primer lugar se utiliza *fuerza bruta* (FB) que consiste en enumerar todas las soluciones posibles, de manera recursiva, y luego buscar entre las soluciones factibles aquella que sea óptima. Para el algoritmo de *backtracking* se introducen podas para reducir el número de nodos del árbol recursivo. Finalmente, se utiliza almacenamiento en memoria para evitar reprocesar resultados de subproblemas ya calculados. Este proceso se conoce como *memoización* y el algoritmo resultante es el de *programación dinámica* (PD).

En la sección de metodología se introducen y explican los algoritmos de cada una de las técnicas utilizadas en el trabajo junto con las respectivas demostraciones de correctitud y complejidad. Luego, se exponen los experimentos realizados con sus resultados y la respectiva discusión. Por último, se detallan las conclusiones finales del trabajo.

2. Metodología

2.1. Fuerza Bruta

La técnica de **fuerza bruta** consiste en recorrer **todo** el espacio de soluciones en busca de aquellas factibles u óptimas. En este caso, el conjunto de soluciones está compuesto por todas las subsecuencias de L. Por ejemplo, si $L = [1, 2, 3]$ con $b = [20, 15, 30]$, $c = [10, 15, 20]$ y $M = 30$ todas las subsecuencias posiblen son $[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]$ y las soluciones factibles son $[1], [2], [3], [1, 3]$.

La idea del algoritmo 1 es generar de forma recursiva las soluciones recorriendo todos los locales, pudiendo elegir incluir o no cada local en cada paso. Al contar con todo el espacio de soluciones, se recorre cada una para determinar si es factible y, en caso de serlo, se devuelve su beneficio total. Luego se selecciona alguna de las óptimas dentro del conjunto de soluciones factibles.

En la Fig. 1 se puede observar un ejemplo del árbol de recursión generado para la instancia detallada más arriba. Cada nodo intermedio representa una solución parcial, es decir, la solución al problema considerando hasta el local i-ésimo. Las hojas representan todas las soluciones posibles. La solución óptima se colorea con verde, las demás soluciones

factibles, con azul y las soluciones no factibles, con rojo. La solución al problema analizado se obtiene llamando a $FB(L, n, M, vecindad, 0)$ donde *vecindad* es un vector de *boolean* inicializado en cada posición en *False*.

La correctitud del algoritmo proviene del hecho de que se recorre *todo* el espacio de soluciones, dado que para cada local de L se generan dos ramas, si se considera para la subsecuencia o no. Y al llegar a las hojas se filtran las soluciones que no sean óptimas o factibles.

El algoritmo genera y recorre de forma completa en todos los casos un árbol de backtracking de $O(2^n)$ nodos. En cada uno se realizan operaciones de tiempo constante excepto en las hojas, en las cuales se realizan los chequeos de factibilidad que son $O(n)$, ya que iteran todo el vector de vecinos. Por lo tanto, la complejidad temporal del algoritmo es $O(2^n \times n)$. Se puede observar que frente a cualquier instancia el algoritmo se comportará de igual manera, dado que genera el mismo tipo de árbol con la misma cantidad de nodos, es por ello que el conjunto de instancias del peor caso es igual al del mejor caso.

Algorithm 1 Algoritmo de *Fuerza Bruta* para NPM.

```
1:  function FB(locales, n, M, vecindad, i)
2:      if i = n + 1 then
3:          beneficio ← 0
4:          contagio ← 0
5:          for j in [1..n] do
6:              if j ≠ n ∧ vecindad[j] = True ∧ vecindad[j + 1] = True then   ▷ Hay dos vecinos consecutivos
7:                  return instancia inválida
8:              if petenencia[j] = True then                                  ▷ El local actual pertence
9:                  beneficio ← beneficio + b_j
10:                 contagio ← contagio + c_j
11:             if contagio > M then
12:                 return instancia inválida
13:         return beneficio
14:     else
15:         Usamos (var ← value) como algo que devuelve var modificado por simplicidad.
16:         return max(FB(locales, n, M, (vecindad[i] ← False), i + 1),
                       FB(locales, n, M − c_i, (vecindad[i] ← True), i + 1)
```

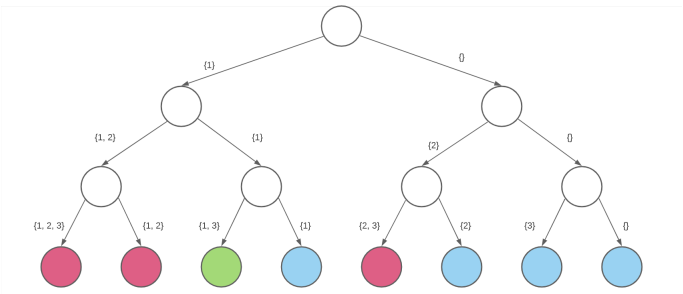


Figura 1: Ejemplo de ejecución del Algoritmo 1 para $L = [1, 2, 3]$ con $b = [20, 15, 30]$, $c = [10, 15, 20]$ y $M = 30$. En rojo las soluciones no factibles, en azul las factibles y en verde la solución óptima $\{1, 3\}$. En cada paso se agrega o no cada local al subconjunto, el cual se marca en las aristas.

2.2. Backtracking

El algoritmo de *backtracking* recorre un árbol para generar las soluciones posibles de manera similar a FB. La diferencia radica en la presencia de *podas*, es decir, reglas que permiten evitar explorar todo el espacio de soluciones omitiendo recorrer ciertas ramas basado en algún criterio. Se dividen en dos tipos: *factibilidad* y *optimalidad*.

- **Factibilidad.** Dada una solución parcial S' representada por un nodo intermedio n_0 que cuenta con un contagio acumulado $m = \sum S'_c$ y el vector de vecindad en estado V'. Como para todo local L_i tenemos valores de contagio

positivos, si $m > M$ entonces no tiene sentido continuar extendiendo S' porque en todos los casos el contagio final excederá M. Además, si en V_i tenemos las últimas dos posiciones del vector indicando que están presentes L_{i+1} y L_{i+2}, cualquiera de las soluciones finales tendrá dos vecinos colindantes y no será factible. En cualquiera de los dos casos, se puede evitar continuar recorriendo el subárbol formado a partir del eje n_0 y así reducir el número de operaciones. Se puede observar el pseudocódigo en el algoritmo 2

- **Optimalidad.** Dada una solución parcial S' representada por un nodo intermedio n_i con beneficio $b = \sum S'_b$. Se cuenta con la solución factible con beneficio máximo B hasta el momento. En cada nodo se calcula b_r, beneficio que se puede obtener al agregar a todos los locales por recorrer. Finalmente, si $b + b_r \leq B$ cualquier decisión que se tome a continuación en el subárbol llevará a una solución al menos tan buena como la ya conocida. Por lo tanto, se puede podar esa rama y evitar cálculos innecesarios. En el algoritmos 2 se actualiza la variable de $maxB$ cada vez que se encuentra una solución factible y se evalúa la regla de la poda. Además, se agrega una pequeña variación que precalcula los valores de b_r para cada nodo y los almacena en una caché. La única diferencia entre ambos es cómo se implementa $maximoBeneficioRestante$ de la línea 8.

Algorithm 2 Algoritmo de *Backtracking* para NPM.

```
1:  function BT(locales, i, M, vecindad, B, maxB)
2:      if M < 0 then
3:          return instancia inválida
4:      if i < (|locales| − 1 ∧ vecinos[i + 1] = True ∧ vecinos[i + 2] = True then
5:          return instancia inválida
6:      if i = 0 then
7:          return B
8:      if B + maximoBeneficioRestante(locales, i) < maxBeneficio then
9:          return instancia inválida
10:     maxLocal ← max(BT(locales, i − 1, M, (vecindad[i] ← False), B, maxB),
                       BT(locales, i − 1, M, (vecindad[i] ← True), B + b_i, maxB)
11:     maxB ← max(maxLocal, maxB)
12:     return maxLocal
```

La solución al problema se obtiene llamando $BT(locales, n, M, vecindad, 0, 0)$. Como en el peor caso no se aplica ninguna poda, se recorren todos los nodos del árbol de soluciones, que ya vimos que son $O(2^n)$. En cada uno se hacen las podas de factibilidad que son de orden constante, y la de optimalidad que es $O(n)$ (ya que se recorren en el peor caso todos los locales) o $O(1)$ si esta cacheada. El resto de las operaciones se hacen en orden constante. Por lo tanto, la complejidad temporal es $O(n \times 2^n)$ y $O(2^n)$ si se usa la variante cacheada.

Existen instancias donde el algoritmo va a recorrer el árbol de soluciones completos, por ejemplo todas las que presenten contagios $c = [k, k, \ldots, k]$, beneficios $b = [l, l, \ldots, l]$ y $M > k * n$ con $k, l \in \mathbb{N}_{\geq 0}$. En este caso una solución óptima es agregar todos los locales de posición impar, pues todos presentan iguales contagio y beneficio y además, el valor de M permite agregar virtualmente todos los locales.

Por otro lado, el mejor caso ocurre cuando la solución óptima se encuentra rápido o cuando se corta rápidamente porque se excede la cota de contagios. Las instancias de tipo $b = [1, \ldots, 1, B]$ con $B > n$ y/o con $c = [C, \ldots, C, C']$ con $C > M$ y $C' < M$. Así se encuentra una solución óptima en la primera rama y luego se realizan las correspondientes podas por optimalidad y/o factibilidad lo que garantiza que ningún otro nodo se ramifique. En estos casos el algoritmo se comportará de forma cuadrática.

2.3. Programación Dinámica

Los algoritmos de *programación dinámica* entran en acción cuando existe superposición entre los subproblemas de un problema recursivo. La idea consiste en almacenar el cálculo del primer llamado a cada subproblema y que, en los subsiguientes llamados, se obtenga el valor *memoizado*. En este caso se define la siguiente función recursiva que resuelve el problema:

$$npm\_pd(i, M) = \begin{cases} -\infty & \text{si } M < 0 \\ 0 & \text{si } i = 0 \\ max\{npm\_pd(i-2, M-c_i) + b_i,\ npm\_pd(i-1, M)\} & \text{sino} \end{cases} \quad (1)$$

Coloquialmente se puede definir a $npm\_pd(i, M)$ como el máximo beneficio de una subsecuencia de locales $[L_i, \ldots, L_n]$ que tenga como límite el contagio M. Se observa fácilmente que $npm\_pd(n, M)$ resuelve NPM ya que representa el máximo beneficio de una subsecuencia de L con límite de contagio M. A continuación, se observa que la recursión efectivamente representa lo que se acaba de enunciar coloquialmente.

Correctitud

a. Si $M < 0$ entonces claramente ninguna subsecuencia va a cumplir la cota de contagio ya que todos los valores son enteros positivos. Así, la respuesta es $npm\_pd(n, M) = -\infty$, el neutro en cuanto a max, lo que hará que no se considere.

b. Si $i = 0$ entonces quiere decir que buscamos la subsecuencia de beneficio máximo dentro de una secuencia de locales vacía. En este caso, como el beneficio de la lista vacía es nulo se tiene que $npm\_pd(n, M) = 0$.

c. En el resto de los casos, se desea buscar una subsecuencia de $L^i = [L_1, \ldots, L_i]$ que tenga beneficio máximo con límite de contagio M. De existir una subsecuencia que cumpla las condiciones, puede tener o no al i-ésimo local. Basta entonces con contemplar al local i-ésimo, y luego seguir buscando soluciones de forma recursiva. Si no lo consideramos, seguimos la búsqueda a partir del local siguiente: $npm\_pd(i-1, m)$. Si consideramos al i-ésimo elemento, el resto de la solución tendrá límite de contagio $M - c_i$, y el beneficio total tendrá también b_i, pero además salteamos al siguiente local, ya que si el i-ésimo abre, el $i+1$ nunca podría estar abierto en una solución factible. Lo que es lo mismo que obtener $npm\_pd(i-2, m-c_i) + b_i$. Por último, la mejor solución será el máximo entre ambas.

Memoización

Se puede ver que la función recursiva 1 toma dos parámetros: $i \in [1, \ldots, n]$ y $m \in [0, \ldots, M]$. Los casos con $i = 0$ y $m < 0$ son casos base y se resuelven de manera *ad-hoc* en tiempo constante. Por lo tanto, la cantidad de posibles combinaciones con la que se puede llamar a la función está determinada por la combinación de ambos. Es así que resulta haber $O(n \times M)$ casos posibles, de manera que si se puede implementar una memoria que recuerde y almacene el resultado de los casos ya resueltos se puede calcular una sola vez cada uno y asegurar, de esa manera, resolver el problema sin resolver más de $O(n \times M)$ subproblemas. El algoritmo 3 muestra la idea aplicada a la función 1.

Algorithm 3 Algoritmo de *Programacion dinamica* para NPM.

```
1: function PD(i, M, locales, mem)
2:     if M < 0 then
3:         return −∞
4:     if i = 0 then
5:         return 0
6:     if mem[i][M] no está definido then
7:         mem[i][M] ← max{PD(i − 1, M, locales, mem),
                           PD(i − 2, M − c_i, locales, mem) + b_i}
8:     return mem[i][M]
```

La complejidad del algoritmo queda determinada por la cantidad de posibles estados o subproblemas que se resuelven y el costo de resolver cada uno. Como se mencionó anteriormente, se resuelven a lo sumo $O(n \times M)$ subproblemas, y cada uno en tiempo constante. Así es como se deduce que el algoritmo tiene complejidad $O(n \times M)$ en el peor caso. La memoria se puede implementar como una matriz de acceso y escritura constante y su inicialización tiene un costo $O(n \times M)$, es por eso que tanto el peor como mejor caso tendrán complejidad $O(n \times M)$.

3. Experimentación

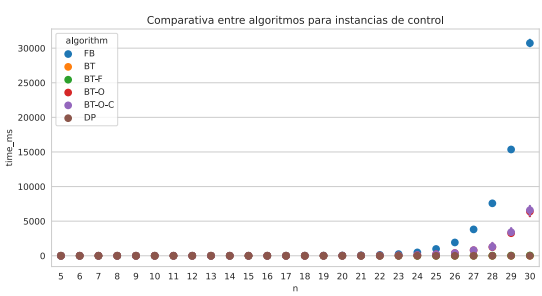
En la experimentación se intentará mostrar las fortalezas y debilidades de cada técnica, comparándolas entre sí para diferentes tipos de instancias. Además, cada instancia tendrá una cantidad de locales $n \in [1, \ldots, 30]$, lo que permitirá apreciar la evolución del tiempo de ejecución en función de n, y compararlo con la complejidad teórica descrita en la Sección 2. Las ejecuciones fueron realizadas utilizando el lenguaje de programación $C++$, en una computadora con un Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz (32K cache lvl 1, 256K lvl 2 y 6MB lvl 3) y 16GB de RAM.

3.1. Métodos

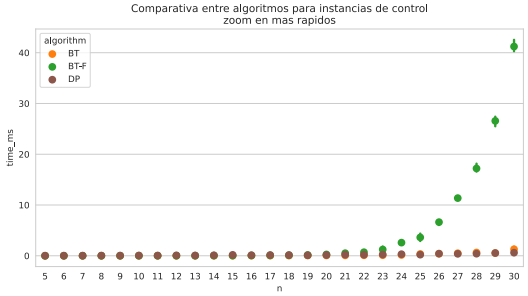
Las técnicas a considerar son las siguientes:

- **FB**: algoritmo 1 de Fuerza Bruta, $O(n \times 2^n)$
- **BT**: algoritmo 2 de Backtracking con ambas podas (optimalidad cacheada y factibilidad). $O(2^n)$
- **BT-O**: algoritmo de Backtracking solo con podas de optimalidad, con factibilidad en las hojas para todos los locales. $O(n \times 2^n)$
- **BT-O-C**: algoritmo de Backtracking con podas de optimalidad cacheadas, se calculan antes de iniciarlo, pero las podas de factibilidad se hacen en las hojas recorriendo todos los locales. luego su complejidad será $O(n + n \times 2^n) \subseteq O(n \times 2^n)$
- **BT-F**: algoritmo de Backtracking solo con podas de factibilidad, por lo tanto su complejidad será $O(2^n)$
- **DP**: algoritmo 3 de Programación Dinámica, $O(n \times M)$

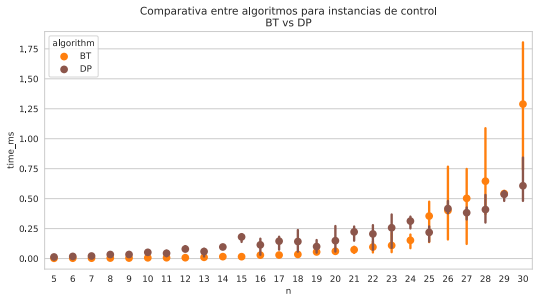
3.2. Control



(a) Entre todos los algoritmos para cada n, con instancias de control



(b) Zoom en los algoritmos más rápidos para ver las diferencias



(c) BT vs DP

Figura 2: Comparativa para dataset de control

Las instancias de *control* se generan de manera uniformemente aleatoria a partir de un rango de beneficios y contagios de $[10, 100]$, mientras que el límite de contagio M se obtiene aleatoriamente a partir del rango $[10 \times n, 100 \times n]$ donde n

es la cantidad de locales. Estos grupos de control permiten comparar las instancias buenas y malas con casos *promedio* ya que se generan todas bajo las mismas condiciones.

Como primera comparación de los algoritmos, en la Figura 2 se puede ver a grandes rasgos la *performance* de cada técnica. Se observa que BT y DP son los que presentan mejores tiempos de ejecución. Lo que puede resultar llamativo es que, viendo la Figura 2c, a pesar de que BT tenga mayor variación, no parecen mostrar grandes diferencias en tiempos de ejecución. Esto puede deberse a que para el caso general, lo que suele definir si una técnica tiene un buen desempeño es si poda por factibilidad o no. Para el caso de las instancias de control, los tiempos observados en BT-F se deben a que es probable que se tome un M que permita introducir a todos los locales y en algunos casos se pierde el efecto de las podas.

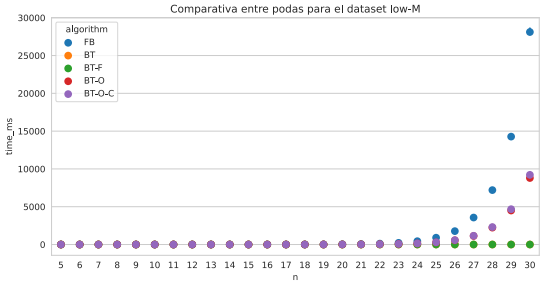
3.3. Podas

Se estudiarán ambas podas por separado: factibilidad y optimalidad. Se compararán los resultados con FB a modo de control. Es importante notar que no es posible ver un caso *malo* para la poda de factibilidad, ya que se hace en $O(1)$.

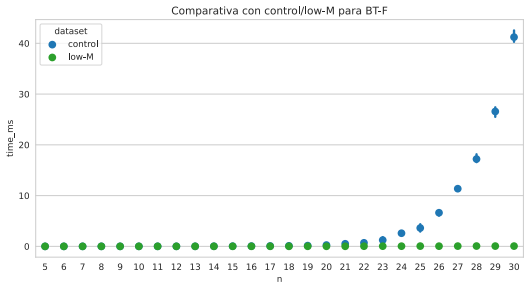
3.3.1. Factibilidad - LowM

Una instancia de *LowM* se construye tomando de forma uniformemente aleatoria valores de beneficio dentro del rango $[10, 50]$, contagio dentro de $[50, 100]$ y un M fijo de 200. De esta manera, se espera que rápidamente se corte la ejecución por estar excediendo el límite de contagio, situación que se alcanzará con pocos locales.

Como se puede ver en la Figura 3a, aquellos algoritmos que presentan poda por factibilidad son más rápidos para este tipo de instancias. Además, en la Figura 3b se ve la comparación con el control, lo que muestra que es un caso más rápido que el promedio.



(a) Comparativa entre podas. Todos los valores de BT y BT-F aparecen solapados.



(b) Comparativa entre los datos de control y low-M para BT-F. Notar que los tiempos de ejecución se encuentran tres órdenes de magnitud abajo de los observados para FB.

Figura 3: Comparativa para instancias de *lowM* entre distintos algoritmos (3a) y solo para BT-F contra el *control* (3b).

3.3.2. Optimalidad

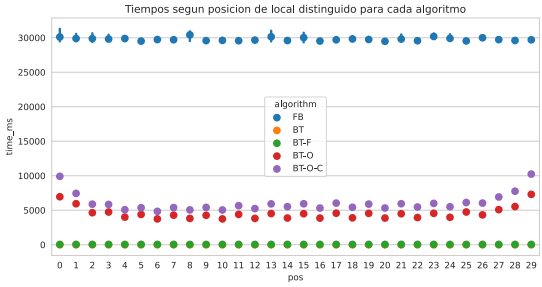
La poda por optimalidad realiza recortes en instancias que, observando los locales restantes, no pueden llegar a obtener un mejor resultado que el máximo resultado obtenido hasta el momento. Se proponen entonces dos clases de instancias distintas. Ambas situaciones no serán un caso común dado un grupo de locales aleatorios, pero es interesante analizar su comportamiento dado que aún así es una situación posible.

- **Uniformidad**: en caso de que todos los locales presenten el mismo beneficio, no van a existir ciertas ramas que produzcan mejores resultados que otras. En estas circunstancias, nunca se aprovechará la poda por optimalidad, pero siempre se estará pagando el *overhead* de computar el beneficio restante.
- **Uno para dominarlos a todos**: en estas instancias existe un local distinguido, que tiene beneficio mayor a la suma de todos los demás. Esto se espera que propicie el uso de la poda por optimalidad.

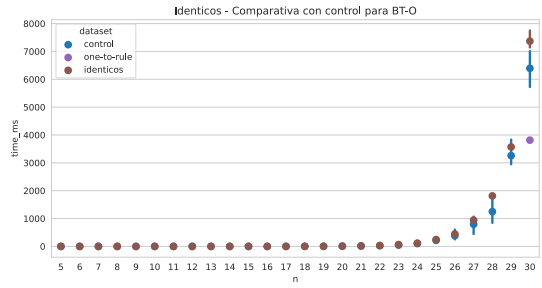
La posición del local distinguido determina la relevancia de la poda. Si se encuentra primero, su beneficio se va a tener en cuenta luego de haber visto todas las hojas, por lo que va a ser lo mismo que el caso uniforme. En cambio, si está del tercero en adelante, se debería almacenar rápidamente su beneficio, haciendo que se poden el resto de los casos. Por lo tanto, se correrá solo para $n = 30$ variando todas las posiciones posibles.

Se espera que la implementación *cacheada* presente mejores tiempos de ejecución respecto de la que lo computa en el momento.

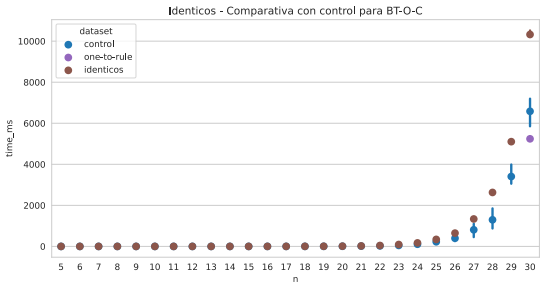
(a) Comparativa de tiempos de ejecución de podas para el dataset de idénticos. Se observa solapamiento para los casos de BT-F y BT.



(b) Tiempo por posición de comercio distinguido en one-to-rule. Se observa solapamiento para los casos de BT-F y BT.

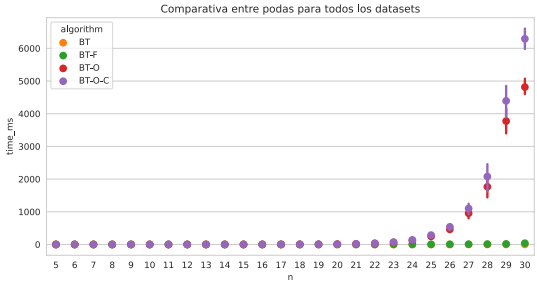


(c) Comparativa con control para BT-O

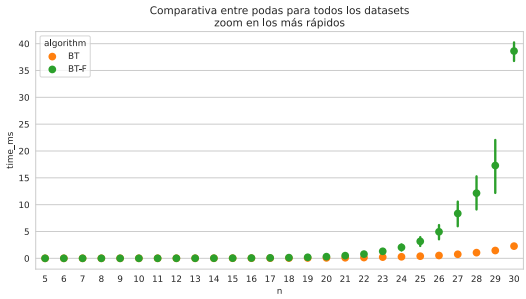


(d) Comparativa con control para BT-O-C

Figura 4: Optimalidad: one-to-rule e indenticos



(a) Comparativa de tiempos para podas.



(b) Zoom para poder diferenciar el solapamiento para los casos de BT-F y BT.

Figura 5: Comparativa de tiempos para todas las podas y todos los datasets.

A pesar de ser un caso favorable, Fig 4a muestra que el hecho de no estar aplicando podas por factibilidad hace que aquellos algoritmos que solo aplican podas por optimalidad sean peores que BT y BT-F, pero al menos siguen siendo mejor que FB. Además, en 4b vemos como según la posición en la que aparezca el local distinguido, cambia el tiempo de ejecución de la misma manera para BT-O y BT-O-C y, como era de esperar, no cambia para los demás algoritmos que se mantienen relativamente constantes. Finalmente, al comparar con el control se observa lo esperado, tiene un tiempo de ejecución peor al del mejor caso (one to rule), pero mejor que el peor (idénticos).

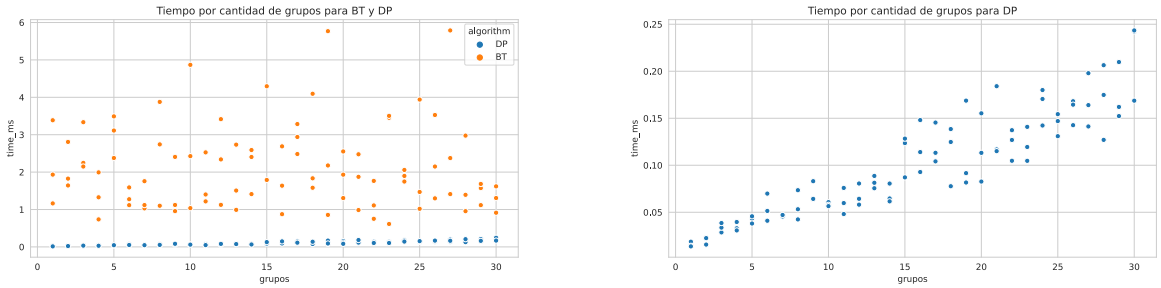
Por último, se observa en Fig 5 una comparación de todos los algoritmos. De aquí surgen dos observaciones interesantes. En primer lugar, contrario a lo esperado, la variante con caché de la poda de optimalidad termina siendo más lenta que la que simplemente lo calcula en cada nodo intermedio. Esto puede deberse a que para valores chicos de n, como los utilizados en este trabajo, el costo de ir a memoria es mayor que el requerido para recorrer el vector de locales. El vector de locales es accedido desde varias regiones de código, mientras que los valores precalculados sólo se utilizan en la poda. Esto podría llevar a su desalojo de la *caché* nivel 1 del CPU, teniendo entonces que buscarla en RAM, lo que presenta un mayor costo temporal. Esto no debería ocurrir con valores de n tendiendo a infinito.

Por otro lado, también puede verse que el mejor de todos fue BT como era esperable y, si bien las podas de optimalidad no le brindaron mucha mejora con respecto a BT-F, se nota cada vez más a medida que aumenta el n.

3.4. Grupos - Solapamiento

Al margen del solapamiento que se da por la naturaleza del problema, se pueden construir instancias que hagan que sea aún mayor. Intuitivamente, a mayor solapamiento más valores van a estar memoizados, y por lo tanto podremos podar más ramas, así reducir el tiempo de ejecución.

Se llama *grupo* a un conjunto de locales con el mismo nivel de contagio. Por ejemplo, dado un $c = [1, 1, 3, 1, 1, 3]$ se observan dos grupos, uno con contagio 1 y otro con contagio 3. A mayor cantidad de grupos, menor será el solapamiento de los subproblemas a calcular, lo que provocará un peor desempeño del algoritmo PD, dado que necesitará calcular una mayor cantidad de entradas de la tabla. Este comportamiento, sin embargo, no se espera que afecte al algoritmo de BT.



(a) Comparativa de tiempos para cantidad de grupos

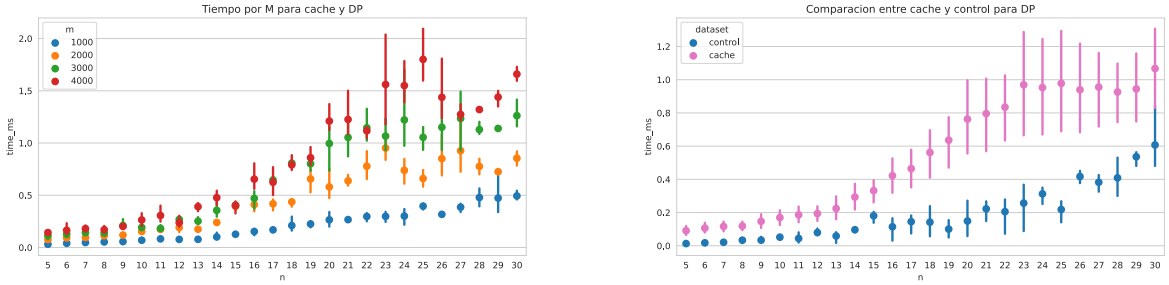
(b) Zoom en solo DP

Figura 6: Grupos

Como se puede ver en Fig 6, los grupos no afectan tanto a BT, pero a mayor cantidad de grupos mayor tiempo de ejecución para DP, como era esperado.

3.5. Caché

Para ver el lado negativo de la memoización, si se cuenta con una estructura lo suficientemente grande como para que no entre toda en caché, para instancias en las que no hay solapamiento debería notarse el *overhead* de ir a memoria.



(a) Tiempo de DP para cache por cada M

(b) Comparativa con dataset control

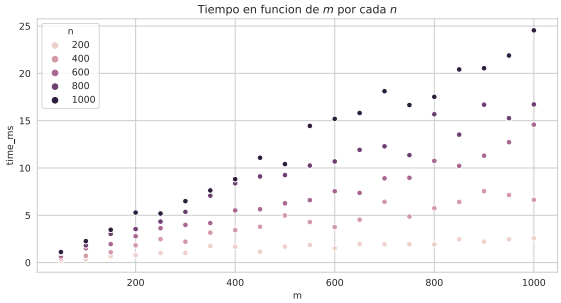
Efectivamente, a mayor M, mayor el tamaño de la estructura de memoización y, por lo tanto, mayor el tiempo de ejecución. Comparándolo con el *dataset control* se aprecia que es peor que el caso promedio.

3.6. Tiempos vs Complejidad teórica

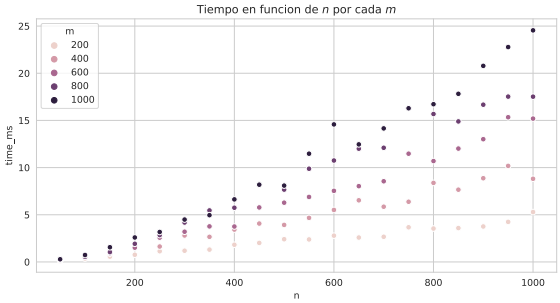
Para verificar experimentalmente las complejidades propuestas, se corren los algoritmos contra instancias de diferentes tamaños y se comparan sus tiempos de ejecución con la complejidad teórica. Luego se calcula el grado de correlación mediante el coeficiente de Pearson.

3.6.1. DP

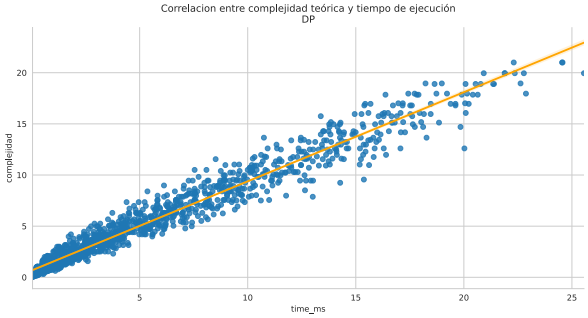
En este caso no es tan simple como para los demás, ya que la complejidad depende de dos parámetros en lugar de sólo uno. Por eso se graficó cada una por separado en función de la otra. En todos los casos se debería tener una relación lineal.



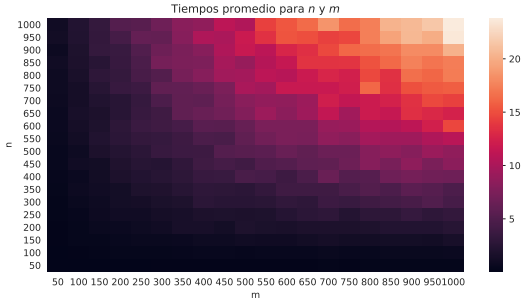
(a) Tiempo en función de m para cada n



(b) Tiempo en función de n para cada m



(c) Comparación con $O(n \times M)$, Pearson: 0.978

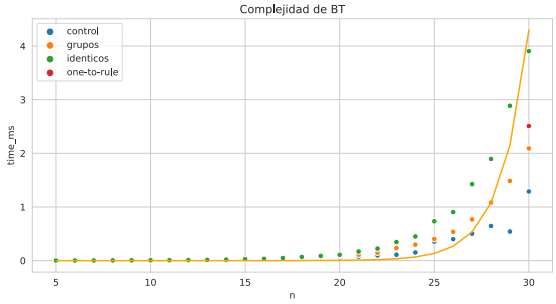


(d) Tiempos para cada n y m

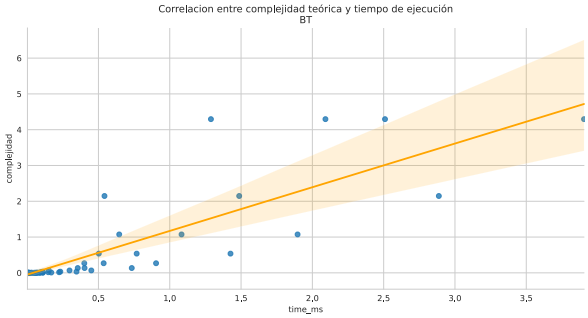
Figura 8: Tiempos para DP

Efectivamente, como se puede ver en 8a y 8b hay una relación lineal entre n y m, la cual esta fuertemente correlacionada (0,978) con la complejidad teórica. Otra forma de visualizarlo es con el *heatmap*.

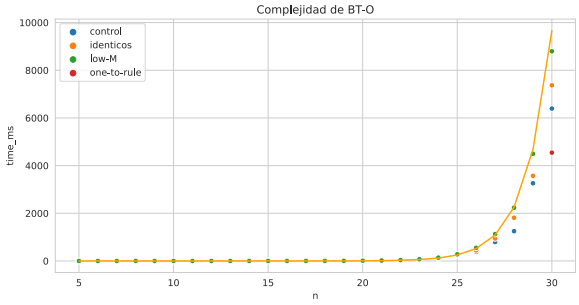
3.6.2. BT



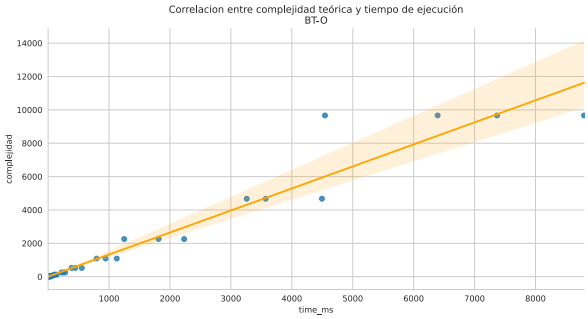
(a) BT - Tiempos vs $O(n \times 2^n)$



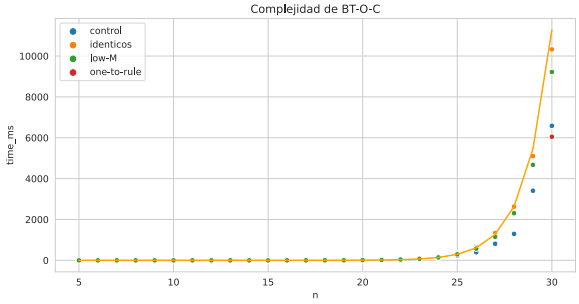
(b) BT - Pearson: 0.828



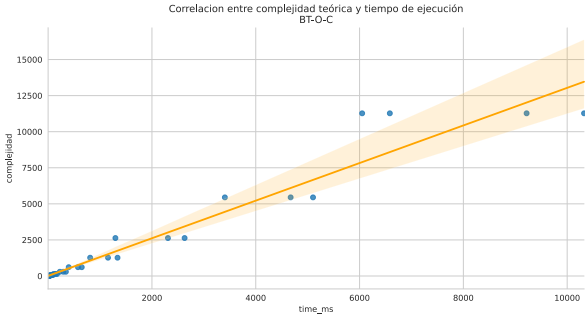
(c) BT-O - Tiempos vs $O(n \times 2^n)$



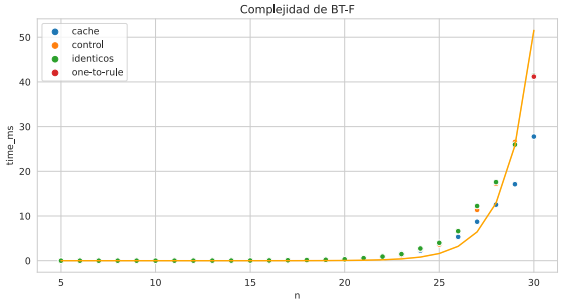
(d) BT-O - Pearson: 0.947



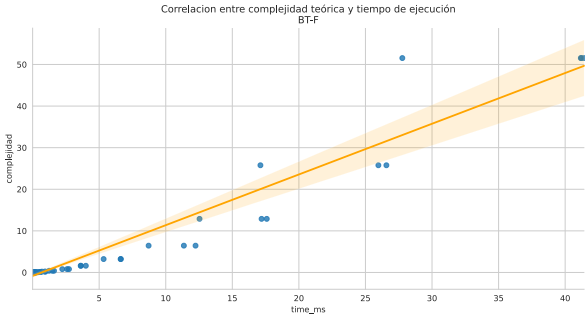
(e) BT-O-C - Tiempos vs $O(n \times 2^n)$



(f) BT-O-C - Pearson: 0.955



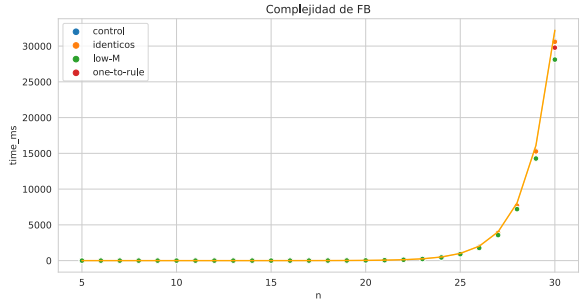
(g) BT-F - Tiempos vs $O(2^n)$



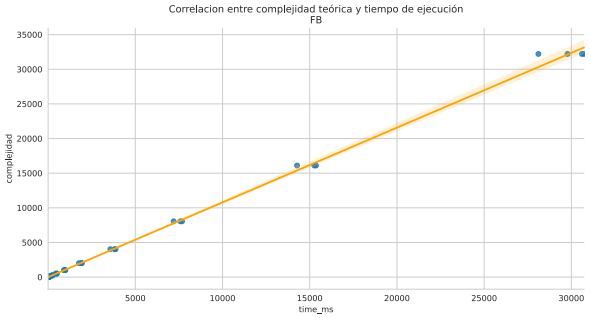
(h) BT-F - Pearson: 0.967

Los tiempos de ejecución de las variantes de BT se asemejan a la curva de la complejidad teórica propuesta. Se obtuvieron buenos valores de los coeficientes de Pearson para cada algoritmo en el rango 0,828 a 0,967.

3.6.3. FB



(a) FB - Tiempos vs $O(2^n)$



(b) FB - Pearson: 0.999

Como se puede ver, al ser el algoritmo que menos varía según el caso, ya que siempre recorre *todo* el espacio de búsqueda, su tiempo de ejecución se asemeja mucho en todos los casos a la complejidad teórica en peor caso. Esto también se ve reflejado en un buen coeficiente de Pearson (0,999).

4. Conclusiones

En este trabajo se presentaron tres técnicas distintas que permiten resolver el problema de NPM, y se comparó su performance evaluándolas para diferentes tipos de instancias que muestran las fortalezas y debilidades de cada una: aleatorias, *one to rule*, según grupos, entre otras. El algoritmo de *Fuerza Bruta* es poco eficiente temporalmente ya que para un número relativamente pequeño de locales ($n = 30$) el tiempo de ejecución se hace excesivamente elevado para todos los casos. Esto se ve mejorado en *Backtracking* mediante el uso de *podas*, donde la máxima performance se alcanza con una combinación de podas por factibilidad y optimalidad, permitiendo reducir el tiempo significativamente, en especial para los tipos de instancias favorables para las podas. Finalmente, el algoritmo de *Programación Dinámica* es el que presenta un menor impacto en su tiempo de ejecución aumentando considerablemente el n, pero lo hace con un gran costo adicional en memoria. Además, pasa a depender del límite de contagio M, lo que hace que para valores elevados no sea la mejor opción.

Una línea de trabajo a futuro es ver si para algún tamaño n de la secuencia de locales empieza a presentar mejores resultados precalcular los valores utilizados en la poda por optimalidad. Además, podrían implementarse otras estructuras para realizar la memoización en el caso de programación dinámica, tal podría ser el caso de mitigar el efecto de matrices ralas (con muchos ceros) que se dan para valores de M grandes, que en la implementación utilizada en este trabajo llevarían a un gran desperdicio de memoria y tiempo de ejecución en su inicialización.

Sería interesante también explorar alguna potencial mejora en la poda por optimilidad utilizada, dado que se decidió usar un algoritmo bastante simple en este trabajo. En lugar de contabilizar todos los locales que falta por recorrer, se podría pensar alguna estrategia golosa que permita reducir los tiempos de ejecución que quizás cobren relevancia frente a instancias más complejas.