DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Negocios por medio

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Luciano Melhem	197/20	lucianomelhem@hotmail.com
Victor Asmad	760/19	vasmad@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

1. Introducción

Este trabajo consiste en resolver el problema de negocios por medio. En el problema se define una secuencia de tamaño n que llamaremos L=$[1, 2, ..., n]$, y dos atributos asociados a cada elemento de esta secuencia que llamaremos beneficio b_i y contagio c_i, naturales. Se quiere encontrar el máximo beneficio. Para esto se eligen elementos de L ($L' \subseteq L$) no consecutivos ($\neg\exists i \in L' : i + 1 \in L'$) sumando sus atributos de beneficio y verificando que la suma de los atributos de contagio no supere un limite M ($\sum_{i \in L'} c_i \leq M$).

Por ejemplo, los contagios y beneficios son b=$[5, 15, 15, 5, 25, 40]$ c=$[50, 20, 20, 10, 15, 10]$,si $M \geq 40$ la solución óptima resulta de elegir el 2do elemento, el 4to y el ultimo, obteniendo un beneficio de 60. No seria posible por ejemplo elegir todos los elementos de la secuencia porque se supera el límite de contagio. Tampoco se podría elegir el 5to si elegimos al 6to, que es el que mas beneficio tiene. No seria óptimo tomar el 1er elemento porque trae mucho contagio, que podría se utilizado para agregar otros. En cambio, si $30 \leq M < 40$ la solución óptima resulta de elegir el 2do o el 3ero y el último, con un beneficio de 55. Notar que existen dos soluciones posibles, pero el resultado es el mismo, el beneficio que se consigue. En este caso M restringe la posibilidad de elegir más locales, es decir, solo se eligen 2 locales cuando con un M suficientemente grande se podrían elegir hasta 3=6/2, en general n/2. Y si $M < 30$ la solución óptima es 65

Para abordar el problema utilizamos 3 técnicas algorítmicas e implementamos con cada una de estas una solución. Las técnicas son Fuerza Bruta, Backtracking y Programación Dinámica. A continuación se explican las ideas que generan cada solución.

2. Fuerza Bruta

Fuerza Bruta recorre todo el espacio de soluciones en búsqueda de aquellas soluciones válidas según el problema. Esto se logra utilizando recursividad. En el problema de negocios por medio el algoritmo de fuerza bruta hace 2 llamados recursivos mientras recorre la secuencia de locales L con un índice i. Un llamado recursivo considera la solución en la que se agrega el elemento i-ésimo de la secuencia, con su beneficio y contagio. El otro llamado recursivo considera la solución en la que no se agrega el elemento i-ésimo. El caso base se alcanza cuando el índice i completa el recorrido de L. Una solución válida de una invalida se distingue en el caso base. Si la suma de contagios de esa solución no supera M, es válida. Y para encontrar la solución que lleva al beneficio máximo, entre todas se calcula el máximo de beneficio al finalizar el recorrido.

Algorithm 1 Algoritmo de Fuerza Bruta para Negocios por Medio.

```
1: gananciaFinal ← 0
2: function FB(locales, M, i, gananciaDeSolucion, contagioDeSolucion)
3:     if i = |locales| then
4:         if contagioDeSolucion < M ∧ gananciaDeSolucion > gananciaFinal then
5:             actualizarBeneficio en gananciaFinal
6:     FB(locales, M, i+1, gananciaDeSolucion + b_i, contagioDeSolucion + c_i)
7:     FB(locales, M, i+1, gananciaDeSolucion, contagioDeSolucion)
```

El problema se resuelve correctamente porque se recorren todas las soluciones posibles, sin saltearse ninguna. A medida que se va recorriendo la secuencia de elementos se generan llamados recursivos que terminan en casos base. Todos los casos base son las posibles soluciones al problema. Y las que verifican la restricción de M entre estas posibles soluciones, que consideramos soluciones válidas, efectivamente actualizan el beneficio que el algoritmo devuelve, resultando en el máximo entre todos los beneficios de las soluciones válidas. Este máximo es el beneficio óptimo o solución al problema.

En cuanto a la complejidad, salvo los llamados recursivos, todas las operaciones son de tiempo constante acotado, no dependiente del tamaño de entrada (O(1)). Los llamados recursivos se realizan 2^n veces. Esto ocurre porque en cada llamado recursivo se hacen otros 2: el primer llamado

recursivo llama a otros 2, esos 2 llaman a su vez a 2 cada uno, por un total de $4 = 2^n$, y esto se repite n veces. Como cada llamado recursivo hace operaciones O(1), la complejidad del algoritmo queda acotada por O(2^n)*O(1) que es la cantidad de llamados recursivos, por el costo de cada llamado recursivo sin contar la parte recursiva.

3. Backtracking

El problema resuelto con backtracking aprovecha ciertas propiedades del problema para evitar hacer llamados recursivos. En particular, cuando el contagio acumulado entre los elementos, que ya se eligieron para formar parte de la solución parcial, más la contagio del elemento que se esta iterando supera el límite permitido por la solución, no vale la pena considerar las soluciones que se desprenden de agregar ese elemento a la solución parcial. En otras palabras, si agregar el elemento en iteración actual inválida la solución parcial, no es de interés esa opción. Por lo tanto no es necesario invocar al llamado recursivo que se desprende de agregar ese elemento.

Esto es conocido como una poda, y siempre que se hace se dejan de lado soluciones parciales que no satisfacen las restricciones del problema. Esto es debido a la propiedad domino: cuando una solución parcial deja de ser valida, las que se extiende de esta también. Todas las invocaciones de otros llamados recursivos que se hagan con una solución parcial que ya es invalida, también resultaran en soluciones invalidas.

Otra de las podas implementadas no es por factibilidad, es decir, no deja de lado soluciones porque no sean validas, es por optimalidad. Esto significa que las soluciones no se analizan porque se conoce de antemano que hay otra solución que es mejor. Se cuenta con un conjunto C de suma de beneficios. En el primer elemento de S esta el beneficio del primer elemento de L, en el segundo elemento tiene la suma $s_1 + b_2$, el tercero $s_2 + b_3$, y así sucesivamente va acumulando los beneficios hasta que s_n tiene la suma de todos los beneficios. La poda consiste en comparar en el elemento i-ésimo del conjunto S junto sumado a la ganancia acumulada de la solución parcial, que representan la máximo beneficio posible para esa solución, con el mejor beneficio calculado hasta ese momento. Si resulta que la solución parcial no logra superar ese máximo beneficio no se realizan los siguientes llamados recursivos para esa solución parcial. Como el beneficio ya calculado es superior que cualquiera que puede extenderse de la solución parcial, esa solución parcial no produce un beneficio óptimo.

Algorithm 2 Algoritmo de Backtracking para Negocios por Medio.

```
1: gananciaFinal ← 0
2: function BT(locales, M, i, gananciaDeSolucion, contagioDeSolucion, sumaRestante)
3:     if i < n then
4:         if contagioDeSolucion + c_i > M then
5:             BT(locales, M, i + 1, gananciaDeSolucion, contagioDeSolucion)
6:         else if gananciaDeSolucion + s_i > gananciaFinal then
7:             gananciaFinal ← máx(gananciaDeSolucion + b_i, gananciaFinal)
8:             BT(locales, M, i + 2, gananciaDeSolucion + b_i, contagioDeSolucion + c_i)
9:             BT(locales, M, i + 1, gananciaDeSolucion, contagioDeSolucion)
```

Algoritmo 2: Implementación con backtracking con poda por factibilidad y optimalidad (en rojo). La versión sin poda por optimalidad no incluye lo rojo.

A continuación se mencionan 5 razones por las que el algoritmo es correcto:

(1) Todas las soluciones pertenecen al conjunto de soluciones validas, es decir, no se agregan soluciones que sean no validas. Esto significa que, el contagio acumulado nunca supera M y, a su vez, no hay elementos consecutivos.

(2) Todas las podas eliminan soluciones no validas o no óptimas, nunca eliminan una solución que puede ser el resultado buscado. La poda por factibilidad es directo que elimina soluciones no validas, por la propiedad domino. Y la poda por optimalidad tampoco elimina la solución

correcta, ya que cuando quita llamados recursivos, deja de analizar soluciones que dan un beneficio subóptimo. Entonces si siempre se quitan soluciones que dan un beneficio subóptimo, y se analizan todas las demás, se esta analizando la solución con mayor beneficio en todas esas "demás".

(3) Entre todas las soluciones que se podan, y las que se comparan, se forma el conjunto de todas las soluciones al problema, validas e invalidas. Esto es porque de no estar las podas, se cae en el algoritmo de fuerza bruta, que recorre todo el árbol de soluciones.

(4) Las ganancias de las soluciones parciales son menores a las finales, y siempre se pueden extender a al menos una solución valida.

(5) Se sabe que todas las demás soluciones no podadas se comparan, y la que permanece es la de mayor beneficio, osea el resultado correcto para el problema. En otros términos, el algoritmo termina de recorrer todas las soluciones no podadas. El caso base base esta invertido: se sigue recorriendo mientras i no llegue a n. Cuando un llamado tiene un i que llega a n (caso base) se termina la ejecución. Todos los llamados son con un índice i que converge a n, dado que empieza en 0 y se incrementa.

En cuanto a la complejidad del algoritmo, si M es mayor a la suma de contagios la complejidad es $O(2^n)$ con ambas podas, o una sola, dado que las podas pueden, en peor caso, no restringir la cantidad de llamados recursivos. Recordemos que la complejidad esta dada por la cantidad de llamados recursivos multiplicado por su costo. Si M es inferior a la suma, la complejidad es la misma pero en la practica va a demorar menos, ya que se poda. Si M es cero, se recorre se hace siempre un llamado recursivo, y esto tiene complejidad O(n). En lo que refiere a la poda por optimalidad, el algoritmo mantiene la complejidad, pero en la practica demora menos porque se espera que pode. El algoritmo con la poda de optimalidad también tiene un recorrido al conjunto de elementos con complejidad O(n), pero esto no influye en la complejidad final.

4. Programación Dinámica

El problema presentado, resuelto mediante backtracking requiere resolver $O(2^n)$ subproblemas, que pueden variar en cantidad según que podas se implementen y que instancias se quieran resolver. Pero se quieren resolver una cantidad mucho mas pequeña de problemas distintos. Esto es, si se diferencia a cada problema por el elemento en L que se esta iterando y el contagio acumulado en la solución parcial, por cada n índices iterados hay un M que se puede combinar con estos para formar un subproblema diferente. Luego hay $O(n * M)$ subproblemas distintos. Acá se verifica la propiedad de superposición de subproblemas, ya que hay muchos mas llamados recursivos que instancias diferentes. Como se verifica esta propiedad, para mejorar la complejidad del algoritmo de backtracking se propone utilizar programación dinámica.

La implementación de programación dinámica es top-down, por lo tanto solo se trata de guardar todos los resultados de los llamados recursivos o subproblemas diferentes en una matriz M. Y cuando se necesita este valor nuevamente se accede a esta matriz. Esta matriz se denomina estructura de memoizacion. El comportamiento del algoritmo que resuelve el problema queda definido por la siguiente función recursiva:

$$f(i,c) = \begin{cases} 0 & \text{si } i = n, \\ max(f(i+1,c), b_i + f(i+2, c-c_i)) & \text{si } c \geq c_i \wedge M_{ic} = \bot, \\ f(i+1,c) & \text{si } c < c_i \wedge M_{ic} = \bot, \\ M_{ic} & \text{si } \neg M_{ic} = \bot \end{cases} \tag{1}$$

A continuación también se provee de un pseudocódigo de la implementación del algoritmo de programación dinámica:

El algoritmo recorre el mismo árbol de recesión que sus versiones en fuerza bruta y backtracking. En este caso lo que cambia es como se almacena el resultado. En backtracking utilizamos una variable global y en programación dinámica se compara la salida del algoritmo, que contiene el beneficio, entre todos los beneficios parciales. En esencia esta haciendo la misma poda de factibilidad, los llamados recursivos también son los mismos y la comparación entre soluciones validas para

Algorithm 3 Algoritmo de Programación Dinámica para SSP.

```
1: M_ir ← ⊥ for i ∈ [1, n], m ∈ [0, M].
2: function DP(i, r)
3:     if i ≥ |locales| then return 0
4:     if M_ir = ⊥ then
5:         if r ≥ c_i then M_ir ← max(PD(i + 1, r), PD(i + 2, r − c_i)) else M_ir ← PD(i + 1, r)
6:     return M_ir
```

obtener la solución óptima también es buscando el máximo beneficio. Y como la implementación de backtracking es correcta, esta también lo es.

La cantidad de llamados recursivos esta acotada por $n * m$ que es la cantidad de subproblemas distintos. Cada subproblema se puede resolver en distinto tiempo. Una vez esta resuelto y se tiene el dato en la matriz de memoizacion, se obtiene en tiempo O(1). Solo se van a solucionar máximo $n * m$ subproblemas, la complejidad es O($n * m$).

5. Experimentación

Para comparar los 3 métodos se crearon una serie de instancias o entradas para el algoritmo con ciertas características para M o para la distribución de contagios. En cuanto a n, varia de 20 a 30. Dichas características de cada instancia se enumeran a continuación:

- $ganancia_final$: M es la suma de los contagios, y en el 10 % superior de L tiene elementos con contagio mas elevado (todo elemento de la segunda parte es al menos 2 veces el promedio de la parte inferior mas elevado).
- random: M son números aleatorios de 0 a 30, mientras que los contagios son múltiplos de 5 en rango de 5 a 50. M tiende en su valor máximo, es aproximadamente la suma de los contagios /50.
- msumac: M es igual a la suma de los contagios de todos los elementos de L
- msumacsobre2: M es la suma /2
- msumacsobre4: M es la suma /4
- mnorestringe: M es superior a la suma de los contagios entre todos los negocios, por lo tanto, para este caso la complejidad cae en la cantidad de negocios que haya en L

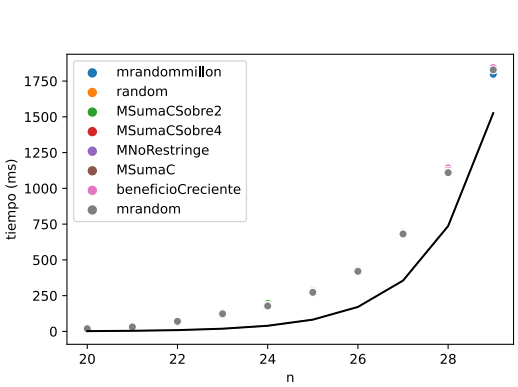
Cada instancia se ejecuta con cada uno de los algoritmos ya mencionados, considerando que backtracking tiene dos implementaciones, una que lleva la poda de factibilidad y otra que, incluye además de esta, la poda por optimalidad. A los resultados obtenidos se les quitaron outliers, tiempos de ejecución que estuvieran por encima del percentil 0.95 y por debajo del percentil 0.05, en todos los casos, con el objetivo de aislar la experimentación del ruido del sistema en el que corre y de posibles fluctuaciones en el throughput.

5.1. Experimento 1: Complejidad de Fuerza Bruta (FB)

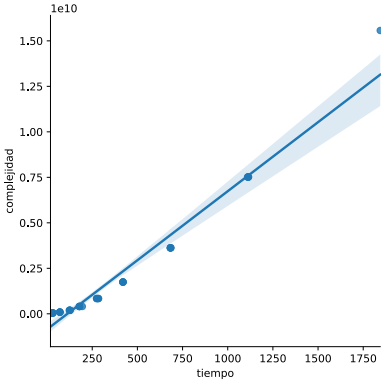
Para empezar con este experimento, se realiza una ejecución del algoritmo de FB evaluando en todas las instancias generadas, repitiendo la ejecución 5 veces para cada n de esa forma se puede obtener una visualización de su tiempo de ejecución. Esta representación se puede ver en la figura 1a. En este gráfico también se superpone la gráfica de una curva $k * 2^n$, con un k fijado visualmente en de manera que se acerca lo mas posible a todos los puntos del los puntos que indican cuanto demoro el algoritmo en cada instancia.

Se evidencia que la curva es muy similar, y esto sugiere que la complejidad es O(2^n) como se esperaba. Para verificar este resultado, también se grafica y calcula la correlación de Pearson en la figura 1b. Del gráfico se observa que los puntos siguen la tendencia lineal esperada para asegurar que la complejidad teórica 2^n se relaciona con el tiempo de ejecución, es decir, que depende una

variable depende de la otra. En otras palabras, que hay un k tal que $k * complejidadTeorica$ se aproxima al $tiempoEjecucion$. La constante k es justamente esa constante que aproximamos visualmente en el gráfico 1a. El coeficiente 0.98 arrojado en la experimentación reafirma esto.



(a) Tiempo de ejecución del método FB sobre todas las instancias.



(b) Correlación de pearson entre el tiempo de ejecución de la instancia mrandom con el algoritmo FB y 2^n.

Figura 1: Implementación FB

Aclaración: Esta ultima verificación es bajo una entrada con M aleatorio entre 0 y un M limite a partir del que, si aumenta, no libera elementos de L para que sean agregados. El valor limite elegido es la suma de los contagios, ya que M impone un valor máximo sobre la suma de estos, y de ser la suma, ya no impone ninguna restricción. Sin embargo, dado que hay una restricción de no agregar elementos consecutivos, el limite real es la suma menos el contagio de la mitad de los elementos de L con menor contagio. Por motivos de simplicidad se eligió el limite como la suma.

Como era de esperar, la variación de M no influye en el tiempo de ejecución del algoritmo de fuerza bruta. Esto ocurre porque fuerza bruta recorre todas las posibles soluciones sin importar M. Es decir, carece de una poda por factibilidad. En el algoritmo de FB los llamados recursivos se realizan siempre, y M solo influye en la actualización del beneficio, que no incrementa la complejidad asintótica porque cada llamado recursivo mantiene su complejidad en O(1) si la rama del if en la linea 4 da true o no (ver algoritmo 1). De no ser así (de no ocurrir que M no influye en el tiempo de ejecución), habrían tiempos de ejecución variables para un mismo n en la figura 1a, porque las instancias graficadas llevan un M variable: M aleatorio, M=suma(contagios),M=suma(contagios)/2,M=suma(contagios)*2,etc. Y se ve que esto no sucede. Por lo tanto M no influye en el tiempo de ejecución en el algoritmo de FB.

5.2. Experimento 2: Complejidad de Backtracking (BT)

Se espera que la complejidad del algoritmo de BT sea exponencial, aunque las podas mejoren el tiempo, pero no se conoce exactamente una métrica que indique cuanto. Se espera que mientras mas podas hayan, menos va a demorar el algoritmo en resolver el problema. No obstante, puede ocurrir que hayan casos particulares en los que un algoritmo con menos podas requiera el mismo tiempo, o incluso mas. Por ejemplo, en esta implementación en particular, se espera que si M=0, la poda de factibilidad se aplique siempre, el algoritmo tenga complejidad O(n), y la poda por optimalidad no pode nunca ya que no hay un beneficio anterior con el cual comparar si se recorre solo una vez. De esta manera, en este caso particular, la implementación con la poda de optimalidad tiene mayor complejidad que la implementación sin: crear el vector de suma de contagios es $O(n^2) > O(n)$.

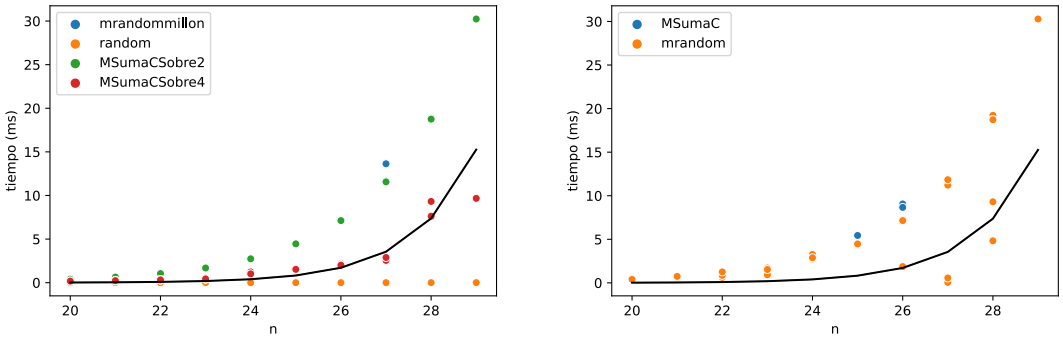
Igualmente, lo mencionado anteriormente, en términos generales no debería suceder. Si se activa una poda, el algoritmo debería demorar menos, acotado superiormente por $O(2^n)$.

En primer instancia se analiza BT sin la poda de optimalidad. En la figura 2b se muestra como con un M aleatorio que restringe, el tiempo de ejecución esta acotado por 2^n pero puede reducirse

en varios casos. Se cree que esta reducción sucede cuando M se acerca a cero, es decir, cuando permite que hayan mas podas. Para corroborar esto, en la figura 2a se puede observar como la poda de factibilidad reduce el tiempo de ejecución en ciertos dataset. Cuando M restringe, las podas se activan y se reduce el tiempo de ejecución. M restringe en las instancias msumacsobre4, random, dado que M vale menos que la suma de contagios. En las demás instancias, no restringe, porque vale mas que la suma, por lo tanto la complejidad se acerca a $O(2^n)$.

También, en la figura 2a, se puede observar como el caso random esta muy por debajo de los demás. Su coeficiente de correlacion con respecto a n es de hecho 0.337. De la instancia MSumaCSobre4 también se calcularon los índices de correlacion de pearson entre el tiempo de ejecución y n (coef=0.801), y entre el tiempo de ejecución y 2^n (coef=0.941). La poda por factibilidad se aplica en varios de los llamados recursivos, reduciendo el coeficiente de correlacion con 2^n en comparación con fuerza bruta, que es cercano a 1. Si bien el coeficiente es alto, no es lo suficientemente cercano. Esto sucede porque las podas se acercan a la complejidad del algoritmo a O(n), pero no llegan a podar todo el árbol si M permite agregar elementos de L a la solución. Los coeficientes con MSumaCSobre2, por ejemplo, son mas elevados con 2^n (coef=0.985) y siguen bajos con n (coef=0.880). Esto muestra que a medida que M restringe, la complejidad de backtracking, mientras menos podas hace, mas se acerca a O(2^n).

Otro aspecto a considerar de los gráficos es que dadas instancias diferentes con iguales n y M los algoritmos suelen demorar lo mismo, es decir, el tiempo de ejecución no parece depender de la distribución de los datos de entrada en el conjunto L.



(a) Los tiempos de ejecución de mrandommillon y msumaCSobre2 se superponen.

(b) Tiempo de ejecución en con las instancias msumaC y mrandom.

Figura 2: Backtracking sin poda de optimalidad, tiempos de ejecución con ciertas instancias de entrada para el algoritmo en función de n. La curva negra es O(2^n).

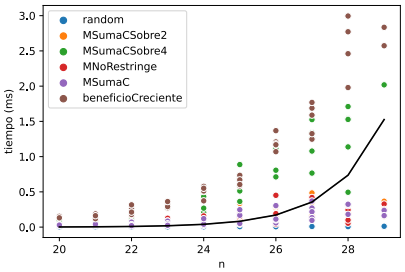
Luego, agregando la poda por optimalidad, en la figura 3a se puede observar como con otra poda mas el algoritmo pasa a ser mucho mas eficiente que sin, dado que el rango de tiempos oscila entre 0 y 5 en vez de entre 0 y 50. Esto se da con todos los dataset. Es decir, hay una mejora global con respecto al algoritmo con una sola poda. De aquí también se observa que cada instancia tiene tiempos de ejecución variable, incluso si M se mantuvo igual entre cada instancia de tamaño n. Este efecto aparece sobre todo en el dataset msumacsobre4 y beneficioCreciente. Esto da un indicio de que la complejidad aquí también depende de la distribución de los elementos en el arreglo.

Un detalle a destacar es que la activación de la poda por optimalidad no anula los efectos de la poda por factibilidad. Instancias con M cercano a cero, como random, siguen teniendo tiempos de ejecución muy bajos.

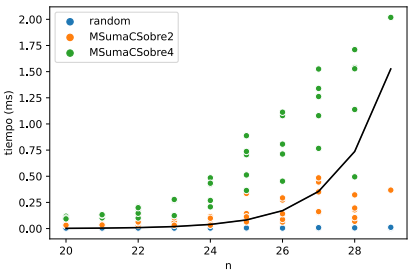
Del gráfico 3b otro aspecto interesante es que un M mas chico no implica que el tiempo de ejecución sea menor. La poda de optimalidad es mas efectiva cuando se agregan mas elementos, pudiendo comparar con una ganancia mayor para podar. Con un M mas elevado, una mayor ganancia se acumula a medida que se avanza en el árbol, y esto permite que la poda de optimalidad

se active de manera mas frecuente. La poda por optimalidad, a diferencia de la poda por factibilidad, no termina de recorrer la rama sin agregar elementos. Esta poda termina la ejecución, excluyendo las soluciones que se desprenden de esa rama.

A partir de estos resultados, se puede destacar que un M mas elevado permite que se active la poda por optimalidad, mientras que M cercano a cero permite que se active la de factibilidad. En caso promedio, con M mayor a cero pero menor a la suma, se espera que se activen las dos podas. La instancia mas representativa de este caso es msumacsobre4. Si bien el algoritmo con las dos podas demora mas que con un M mayor, es mas efectivo que solo con la poda de factibilidad.



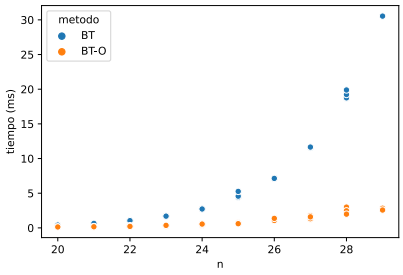
(a) Con instancias que barren distintos M.



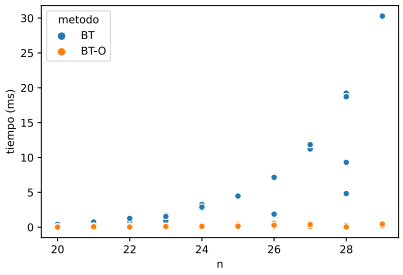
(b) Con instancias MSumaCSobre2 y MSumaCSobre4

Figura 3: Gráfico de dispersión de Backtracking con poda de optimalidad

Por otra parte, comparando las 2 implementaciones, al observar las figuras 3a y 3b la instancia de beneficioCreciente de BT con poda de optimalidad arroja tiempos de ejecución levemente mas cercanos a BT sin podas por optimalidad que con una instancia con distribución aleatoria. Esto se puede apreciar comparando el tiempo de ejecución de BT-O en el rango superior de n entre los 2 gráficos. La causa de la subida en tiempo es que con ganancias mas altas al final del conjunto L la poda por optimalidad se activa mas al final del árbol, recortando menos soluciones y recorriendo mas. Por otro lado, en los 2 gráficos se puede apreciar como, con un m aleatorio entre 0 y la suma, agregar la poda mejora el rendimiento, sin importar que la distribución de las instancias sea desfavorable.



(a) Con instancia beneficioCreciente.



(b) Con instancia mrandom.

Figura 4: Comparación de podas por Backtracking con y sin podas de optimalidad

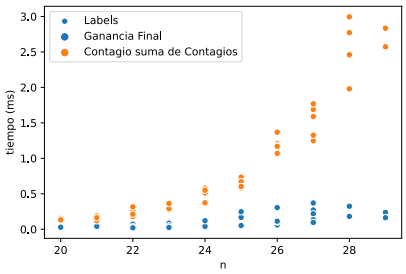


Figura 5: Backtracking con poda de optimalidad, instancias msumac y beneficioCreciente.

5.3. Experimento 3: Complejidad de Programación Dinámica

Con el objetivo de corroborar que la complejidad de la implementación de programación dinámica es $O(n * m)$, se ejecuta el algoritmo 30 veces con un M aleatorio diferente, que se encuentra en el rango $[cero; sumadecontagios]$, con cada instancia. De la instancia random, en la figura 6 cada punto representa el tiempo de ejecución con un color según la escala que aparece en la barra derecha. Se elige M en el mismo rango que n dado si se eligen rangos diferentes los incrementos en tiempos de ejecución pueden no ser representativos en alguna dirección. Si por ejemplo, M=100000, un incremento en n de a 1 no va a influir en el tiempo de ejecución de la misma manera que incrementar 1 M, asumiendo que el tiempo de ejecución tiene complejidad lineal. Del gráfico, la zona en la que n y m son elevados, el tiempo también lo es, y fijando un n o m, al incrementar la otra variable el tiempo de ejecución también aumenta.

En cuanto a los coeficientes de correlacion de las instancias mrandom y random, se obtienen constantes de correlacion 0.94 y 0.89 respectivamente. El coeficiente es mas cercano a 1 con la instancia mrandom. La relación entre el tiempo de ejecución y la complejidad teórica es mas fuerte con M mas elevado (mrandom) dado que agregar muy pocos elementos (random) hace que el algoritmo se comporte de manera lineal con respecto a n. La poda por factibilidad se activa de manera significativa. Sucede algo similar al comportamiento de la complejidad en backtracking. Cuando M es bajo el algoritmo no se ajusta a su cota superior en complejidad. En el caso de backtracking es con 2^n, en dinámica es $n * m$. Estos últimos 2 coeficientes se calcularon con muestreos de mayor cantidad de datos que otros presentados anteriormente, por lo tanto se estima que son mas representativos.

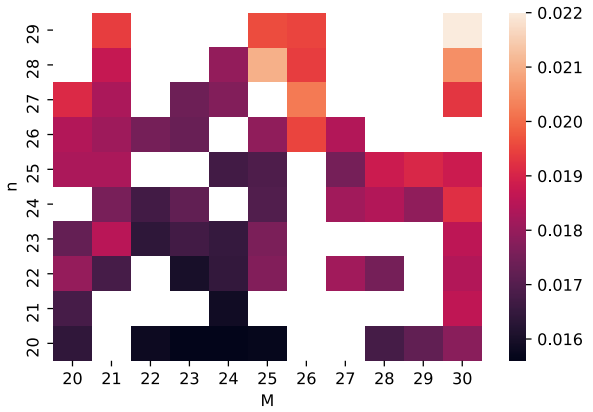


Figura 6: Heatmap de programación dinámica, instancia random. La barra asocia un el tiempo de ejecución en milisegundos con un color.

En backtracking, al incrementar M sobre la suma, el tiempo de ejecución no se ve afectado. A la inversa, en programación dinámica incrementar M sobre la suma sigue afectando la complejidad. Esto ocurre porque se crea una matriz de dimensiones n y m. De no considerar el costo de creación de la matriz, se cree que el algoritmo no tendría un incremento en tiempo de ejecución al elevar M por sobre la suma de contagios. Es decir, se memoizarían los mismos subproblemas salvo que bajo una matriz mas grande y en la región donde el índice correspondiente a m es mas alto. En el gráfico 7 si se considera el costo de creación de la matriz, por lo tanto la complejidad es n*m. Se podría hacer una adaptación del algoritmo que antes de ejecutar calcule la suma de contagios y si M es mayor a este valor, que lo modifique.

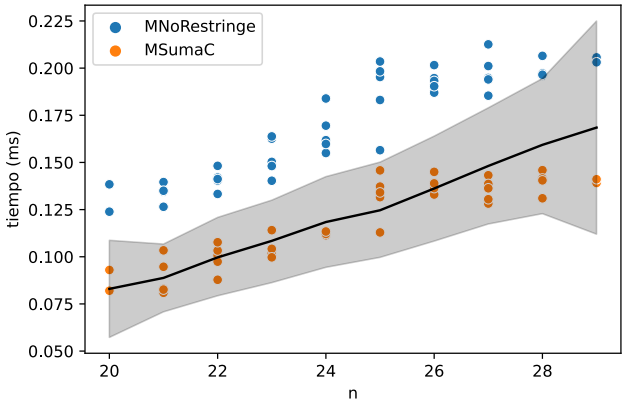
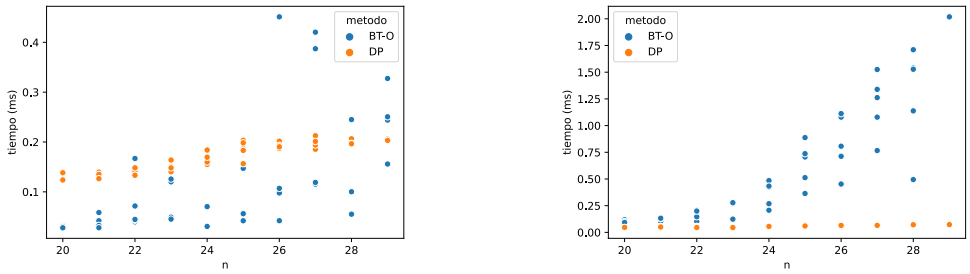


Figura 7: Programación dinámica, instancias mnorestringe y msumac. Se gráfica una recta con el promedio de n*m y una zona que captura el producto $n * m$ máximo y mínimo para cada n. Dado $n = n_o$, el limite inferior corresponde a $n_0 * min(m)$ y el superior $n * min(m)$

Si bien programación dinámica se puede aplicar con podas, las podas no siempre se activan. En otras palabras, no se puede asegurar que el algoritmo sea mejor que O(2^n). Podría suceder, por ejemplo, que un elemento contenga todo el peso de la ganancia, de manera que la poda por optimalidad nunca se active. O a la inversa, que un elemento lleve un contagio muy alto y que además este al final de L. Esto provocaría que la poda de factibilidad no se active. Por estas razones se decide comparar backtracking contra programación dinámica sin podas. En el gráfico 8a BT-O resulta mas eficiente que PD, si bien la cota superior del mas eficiente es exponencial y la del menos eficiente es lineal. Las podas resultan ser muy efectivas, aunque hay algunas ejecuciones que se elevan mas de la cuenta. En el segundo gráfico 8b se puede ver una de las instancias que no llega a activar la poda. Ya con esta combinación de datos de entrada, backtracking no supera en rendimiento PD. Otra instancia comparable con PD es la de beneficioCreciente, que también requiere un tiempo mayor que PD. En ambos casos, los tiempos de ejecución de BT-O esta dispersos si se fija n, aun con un M fijo. No obstante este es el mismo efecto que se puede ver en otros gráficos, solo que la escala en el tiempo es mas precisa y se ve magnificado.



(a) Comparación de backtracking con poda de optimalidad y programación dinámica utilizando el dataset de MNoRestringe

(b) Comparación de backtracking con poda de optimalidad y programación dinámica utilizando el dataset de MSumaCSobre4

Figura 8: Comparación entre técnicas algorítmicas

6. Conclusiones

Para el desarrollo de este trabajo, como bien se mencionó a lo largo del mismo, se utilizaron tres algoritmos que utilizan técnicas distintas para resolver el problema de los Negocios Por Medio

Para empezar, **el algoritmo de fuerza bruta es el menos eficiente** a la hora de resolver el problema puesto que busca todas las soluciones posibles al mismo y, al haber más negocios en la secuencia, incrementa el tiempo de ejecución un nivel exponencial en todos los casos.

Por otro lado, tenemos el algoritmo de Backtracking con sus podas por factibilidad y optimalidad que, como se presenta en su sección correspondiente, dependiendo de la **distribución de los datos de entrada** pueden reducir o incrementar el tiempo de ejecución a la hora de resolver el problema. Puede que ciertas podas no sean tan efectivas como se esperaba, esto se ve reflejado al **cambiar el valor de M y/o el orden de los** b_i. Cuando las podas pierden su efecto, backtracking resulta ser muy similar el complejidad a fuerza bruta, y cuando podan en demasía la complejidad es lineal.

Para finalizar, tenemos **el algoritmo de programación dinámica, que después de todo, resulta ser bastante eficaz** ya que nos permite, para la gran mayoría de distribuciones de L, resolver el problema en tiempos lineales. Inclusive en aquellos casos donde el M es bastante elevado que, mientras en BT pueden afectar significativamente a la complejidad del algoritmo, incluyendo incluyendo su versión con ambas podas, en PD los tiempos de ejecución no varían exponencialmente. No obstante, cuando se habla de **complejidad espacial**, este no es el algoritmo mas ideal para aplicar. Mientras que en BT y FB esta es prácticamente nula, en PD, depende de M y n.

Teniendo estas conclusiones previas sobre cada algoritmo, una futura actividad a tener presente es enfocarnos en la complejidad espacial del algoritmo de Programación Dinámica para mitigar el uso en memoria. También, seria un próximo objetivo, experimentar con otras estructuras de memoización para ver como se comporta este algoritmo.

Otro punto de partida futuro, puede ser poner a prueba distintas distribuciones de datos y analizar el comportamiento de cada algoritmo. Por ejemplo, considerar la instancia de beneficioCreciente con un crecimiento exponencial entre los extremos del conjunto L.