DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Negocios Por Medio

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Gonzalo Martín Bustos	56/19	bustosgonzalom@gmail.com
Santiago Matías Ravaglia	329/19	santi23062000@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

El método NPM (negocio por medio) que plantea el gobierno para reactivar la economía y prevenir una catástrofe epidemiológica consiste en maximizar la ganancia generada por los locales abiertos sin que peligre la población. Para esto, se el asigna a cada local del conjunto de locales L un valor de "contagio" y uno de "beneficio" en base a ciertos parámetros que no son relevantes para este trabajo práctico. Tambien se establece una cota máxima del valor contagio M, la cual define si una combinación de locales abiertos pone en riesgo a los ciudadanos o si se la puede considerar como una potencial solución. Además, aquellas combinaciones de locales que contemplen abrir dos o mas locales ubicados de manera consecutiva, serán descartadas, con el fin de respetar el distanciamiento obligatorio.

Como ejemplo de una solución válida, tomaremos una muestra de n=4 locales, M=30 y las secuencias de beneficios b=[20,15,30,25], c=[15, 5, 20, 10]. En este caso particular, la solución L' que maximiza el beneficio es abrir los locales 1 y 4 tal que L'=[1,4], los cuales suman un total de 45 puntos de beneficio y 25 de contagio. La solución L'=[1,3] que contempla abrir los locales 1 y 3 no es factible ya que a pesar de que provee mas beneficio, tambien supera el limite de contagio M. Por otro lado, la solucion L'=[1,2,4] que contempla abrir los locales 1,2 y 4 tampoco es válida porque los locales 1 y 2 son contiguos.

Para lograr este objetivo diseñaremos tres algoritmos, cada uno demostrando una técnica de diseño distintas con sus respectivas ventajas y desventajas, para luego analizarlos exhaustivamente.

2. Resolución del problema

2.1. Introducción

En esta sección hablaremos de los tres distintos metodos a implementar, denotando por cada uno:

- Razonamiento y funcionamiento de los mismos (con un pseudocódigo de acompañamiento)
- Ventajas y desventajas entre ellos
- Complejidades temporales

Para ésta resolución vamos a tomar como pre-condición para las entradas a los algoritmos que los vectores de locales no pueden ser vacíos.

2.2. Fuerza Bruta

El método de *Fuerza Bruta* se caracteriza por calcular todas las instancias posibles, definir cuales son soluciones válidas y elegir la que, en este caso, maximice el beneficio. Es evidente que con entradas de gran tamaño, la eficiencia de este método decae a gran velocidad (observar análisis de complejidad temporal 2.2), por lo que resolver problemas elaborados con fuerza bruta más allá del poder de cómputo, resulta casi imposible, no porque una computadora no sea capaz de resolver las instancias, sino por el tiempo que conllevaría hacerlo. Para encontrar la mejor instancia de locales abiertos, calculamos recursivamente el conjunto de partes de la secuencia de locales, $\mathcal{P}(L)$. Este cálculo genera 2^n instancias distintas de locales abiertos y cerrados. Como se calculan de forma recursiva eligiendo abrir o no un local, generamos un arbol binario que representa las distintas ramas.º combinaciones posibles, y donde las hojas son todas las soluciones completas (que no necesariamente quiere decir que sean soluciones válidas). En cada iteración en la que se decide abrir un local, como sabemos que no podemos tener locales ubicados de forma contigua, movemos el indice de a dos posiciones en vez de avanzar al siguiente local ya que estariamos evaluando de forma innecesaria una o mas instancias (ésto lo trataremos como una restricción al espacio de soluciones, aclarado más adelante 2.2). Además, en dichas iteraciones se suma al beneficio general acumulado hasta el momento, el valor de beneficio del local sobre el que me encuentro iterando, y lo mismo pasa para el contagio acumulado. Así, cuando llegamos al caso base, corroboramos que el contagio acumulado no supere M y devolvemos el beneficio acumulado.

Algorithm 1 Algoritmo de Fuerza Bruta para el problema del Distanciamiento Social.

```
1: function BF(b, c, M, i, sumB, sumC)
2:     if i >= |b| then
3:         if sumC > M then return 0 else return sumB
4:     return mín{BF(b, c, i + 1, sumB, sumC), BF(b, c, i + 2, sumB + b[i], sumC + c[i])}.
```

La complejidad del peor caso de este algoritmo pertenece a $O(n \times 2^n)$ ya que por cada una de las 2^n instancias generadas por el cálculo del conjunto de partes (es decir, estamos calculando todos los posibles casos pertenecientes al conjunto de partes del conjunto de locales) de L, se recorre la instanca entera cuyo tamaño máximo es n, el largo de L. Para lograr verlo más fácilmente, se puede considerar una secuencia $L = \{l_1, l_2, ..., l_n\}$ de locales, donde nosotros evaluaríamos las soluciones que contienen a l_i con $1 \leq i \leq n$ y aquellas que no contienen a l_i. Por lo tanto estariamos considerando todas las combinaciones posibles, siendo esta cantidad 2^n. Y como recorremos la secuencia de n elementos una sola vez (por la restricción del espacio de soluciones), ésto sucede por cada elemento del arreglo una vez, es decir n veces en total. Por lo que queda de una complejidad de peor caso de $O(n \times 2^n)$.

Cabe destacar que nuestro espacio de soluciones esta medianamente acotado respecto del mas básico en el que en cada caso base se corrobora que la secuencia de locales abiertos generada no contenga locales abiertos de manera consecutiva. Esto se da asi ya que al momento de hacer recursión, si el local sobre el que esta parado $index$ se abre y se suman su beneficio y su contagio a los acumulados, el indice se saltea el siguiente local y pasa directamente al que se encuentra en la posición $index + 2$; luego, en caso de que no se abra el local, el indice avanza una sola posición. De esta manera, nos aseguramos que cualquier instancia que llegue al caso base no tendrá locales abiertos de manera consecutiva y nos ahorraremos el computo de instancias innecesarias.

2.3. Backtracking

El algoritmo de *Backtracking* es similar al algoritmo de fuerza bruta, siendo que el concepto de generar un arbol de todas las posibles decisiones locales y obtener la mejor de ellas se mantiene, pero agregando el concepto de *podas*. Las podas son condiciones que nos permiten cortar aquellas ramas o caminos del arbol donde, gracias a la presencia de éstas condiciones, no tendrían una solución de interés.

Poda por factibilidad: Ésta poda se caracteriza por garantizar que si se cumple su condición, no existe una sucesión correcta (a la larga, una solución válida) al problema propuesto siguiendo el camino sobre el cuál se cumple la condición.

- En nuestro caso, una poda por factibilidad es la siguiente: Sea S_i una solución parcial i tal que su beneficio (suma de beneficios de aquellos locales que hayan extendido a la solución hasta llegar a la instancia S_i) sea b_i, y tal que su contagio (suma de contagios de aquellos locales que hayan extendido a la solución hasta llegar a la instancia S_i) sea c_i, y finalmente, el costo del próximo local a considerar para extender a S_i sea c_j. Sabemos que para que S_i continúe su camino hacia una solución válida, debe cumplir con que el contagio de la solución debe ser menor al límite de contagio M. Por lo que si la suma de c_i con c_j es mayor a M, sabemos que ya no puede extenderse más la solución parcial para llegar a una válida.
  Por ésta misma razón, sabemos que la poda es correcta, ya que siendo los c_k con $0 \leq k \leq \infty$ por definición del problema, no existe c_k tal que $c_i + c_j + c_k <= M$ si $c_i + c_j > M$.

- Otra poda por factibilidad puede ser la segunda restricción propuesta en el enunciado, donde una solución válida es una donde aquellos locales que compongan a S_i no pueden ser contigüos. Es decir, si estamos iterando sobre una secuencia L de n locales, ninguna solución válida puede contener a los locales L_i y L_j con $i \neq j, 1 \leq i, j \leq n$ en el conjunto de locales utilizados para armar la solución S_i. En el caso de nuestro algoritmo propuesto, ésta poda se está cumpliendo de manera ïmplícita", ya que estamos iterando sobre los casos donde justamente no tomamos en cuenta los locales contigüos. Quizás en una implementación diferente, podría verse como una poda, pero en éste caso lo consideraremos como una restricción del espacio de soluciones, tal como fue mencionado en el algoritmo de Fuerza Bruta (2.2).

Poda por optimalidad: Ésta poda se caracteriza por garantizar que si se cumple su condición, no existe una solución mejor a la encontrada siguiendo por esa rama. En nuestro algoritmo, planteamos una poda por optimalidad en la que si el máximo beneficio encontrado es igual al maximo beneficio calculado sin tener en cuenta el limite de contagios, es decir, si en alguna iteración t del algoritmo el máximo beneficio acumulado hasta ese momento (lo llamamos por ejemplo $maxBen_t$) es igual a $S_{max} = max\{\sum_{i=1}^{n} if\ i\%2 == 0\ then\ b_i\ else\ 0$, $\sum_{i=1}^{n} if\ i\%2 == 1\ then\ b_i\ else\ 0\}$, entonces podemos podar esa rama ya que encontramos la solución más óptima.

Algorithm 2 Algoritmo de Backtracking para el problema del Distanciamiento Social

1: **function** $BT(b, c, M, i, sumB, sumC, cotaOpt)$
2: **if** $i \geq length(b)$ **then return** $sumB$
3: **if** $sumC + c[i] > M$ **then return** 0
4: **if** $sumB + b[i] == cotaOpt$ **then return** $sumB + b[i]$
5: **return** $\max\{BT(b, c, i+1, sumB, sumC, cotaOpt), BT(b, c, i+2, sumB + b[i], sumC + c[i], cotaOpt)\}$.

La complejidad del peor caso de éste algoritmo es $O(n^2 \times 2^n)$. Ésto es así porque en el peor caso, no se logra ingresar a ninguna condición de poda, por lo que recorre al igual que el algoritmo de fuerza bruta, todas las posibles combinaciones de locales (dentro del espacio de soluciones restringido propuesto). Ya que $O(n \times 2^n)$ es menor que $O(n^2 \times 2^n)$, por lo que también podemos decir que su complejidad también pertenece a $O(n^2 \times 2^n)$.
Notar tambien que en el algoritmo, cuando se evalúa la condición de la poda, se devuelve 0 en caso de que se cumpla. Ésto es válido ya que, como fue mencionado anteriormente, tantos los valores de los beneficios, como los valores de los contagios de cada local son $k \geq 0$, por lo que el menor beneficio posible es 0.

2.4. Programación Dinámica

Los algoritmos de *Programación Dinámica* son útiles en aquellos casos donde un problema recursivo tiene una *superposición de problemas*. La idea es poder evitar volver a calcular subproblemas ya calculados previamente, utilizando estructuras que almacenen tal información durante la ejecución del algoritmo. En nuestro problema de la Distancia Social, podemos reconocer que tenemos una superposición de problemas a través de un ejemplo como el siguiente: si tuvieramos una secuencia L de 5 locales tal que $L = \{l_1, l_2, l_3, l_4, l_5\}$, y de manera recursiva calculamos el beneficio máximo posible tal como lo pide el enunciado recorriendo desde l_5 hasta l_3. Ahora, si nosotros quisiéramos seguir calculando hasta l_2 o l_1, notamos que la naturaleza recursiva del algoritmo necesitaría recalcular el máximo beneficio entre los locales l_3, l_4, l_5 para el caso donde l_2 ES parte de los locales que consideramos y para el caso donde l_2 NO ES parte de los locales que consideramos. De igual manera con l_1, por lo que notamos claramente la superposición de subproblemas.
En éste caso, el algoritmo propuesto se comporta de la siguiente manera: recorremos la secuencia L de locales desde $l_|L| = n$ hasta l_1, y vamos guardando los resultados calculados en la estructura de memoización elegida. En nuestro caso, la estructura elegida fue una matriz de n ($n = |L|$) por M (límite de contagio máximo), para poder hacer el registro en dependencia de los valores posibles que vaya tomando la suma de contagios, además de la suma de beneficios.

Para mejor entendimiento del algoritmo, mostramos a continuación una función representando el algoritmo de una manera simplificada:

$$f(i,k) = \begin{cases} -1 & \text{si } k < 0, \\ b_i & \text{si } k \geq 0 \wedge i = 0 \wedge c_i \leq k, \\ \max\{f(i-1,k), f(i+1,k-c_i)\} & \text{si } k \geq 0 \wedge i > 0, \\ 0 & \text{caso contrario.} \end{cases} \quad (1)$$

Notar que b y c no estan presentes en esta definicion de f, pero tomamos ésta función únicamente como referencia para entender la solución propuesta junto con sus diferentes partes, reconociendo a b como la secuencia de beneficios de los locales y a c como la secuencia de contagios asignados a ellos. Además de ello, tomar en cuenta de que tampoco se hace mención de nada relacionado a una estructura de memoización ya que no compete a la función matemática, sino a la implementación de la función.

Correctitud

(i) Si $k < 0$ entonces claramente nos indica que excedimos el valor límite de contagio. Ésto sucede porque tomando $k = M$ comenzamos a descontar los valores de contagio de cada c_i, con i el indice de cada local que se recorre en los llamados recursivos, y si en algún punto k pasa a ser menor a 0, hemos restado a M un número r tal que $r \geq M$ y $r \in N_0$. Así es que si $k < 0$ entonces no perdemos soluciones válidas al devolver -1.

(ii) Si no hemos excedido el valor límite de contagio, nos encontramos en el último local a recorrer de la secuencia L de locales (el primero en orden creciente de índice, pero recorremos desde el local n hasta el 1) y el contagio es menor o igual que el límite establecido, entonces quiere decir que estamos en un último local válido para tomar en cuenta en el cálculo, entonces como va a ser el primer valor devuelto de ésta rama recursiva, no hace falta acumularlo a nada, por lo que podemos devolver directamente el valor del beneficio del local. Por lo tanto es correcto devolver $f(i, k) = b_i$.

(iii) Si en cambio estamos en un local válido pero de índice diferente al 1, quiere decir que todavía tenemos camino por recorrer (como fue mencionado en el item (ii)), ya que nos encontramos en un local intermedio entre el comienzo de la secuencia de locales y el primer índice (es decir, estamos en un caso válido tal que el índice que estamos recorriendo $i \in (1, ..., n]$). Por lo tanto, es válido realizar los llamados recursivos de la función, teniendo en cuenta que en caso de **contar** el local actual, tenemos que mantener la resta sobre el límite de contagios (mencionado en item (i)) y que en caso de **no contar** el local actual, solo decrementamos en 1 el índice. Y del resultado de ambos, obtenemos el mayor valor entre ambos beneficios, y lo devolvemos. Por lo tanto es correcto, por definición del problema y por cómo está estructurada la función recursiva.

(iv) En cualquier otro caso, devolvemos el mínimo beneficio posible, por lo que sigue representando una solución válida.

Memoización La estructura de memoización utilizada en la implementación es una matriz de $n \times M$ con $n = |L|$, siendo L la secuencia de locales. Tal como fue mencionado anteriormente (párrafo introductorio de Programación Dinámica 2.4), tomamos n y M para poder almacenar la información necesaria que depende tanto del índice del local como de su nivel de contagio. Por lo que, con cómo está armado el algoritmo, al ir disminuyendo el valor del contagio restante (parámetro k), también estamos marcando una posición diferente dentro de la estructura. Es decir, podemos tener más de un caso donde el índice del local i se tenga en cuenta, gracias a que la matriz también distingue sus posiciones según el k. Para mayor claridad se provee de un pseudocódigo:

Algorithm 3 Algoritmo de Programación Dinámica para el problema del Distanciamiento Social.

```
1: Memo_ik ← ⊥ for i ∈ [1, n], w ∈ [0, W].
2: b ← (secuencia de beneficios de locales)
3: c ← (secuencia de contagios de locales)
4: function DP(i, k)
5:     if k < 0 then return −1
6:     if Memo_ik == ⊥ then
7:         if i == 0 then
8:             if c_i <= k then Memo_ik = b_i
9:             else Memo_ik = 0
10:        else
11:        skip ← DP(i − 1, k)
12:        count ← DP(i − 1, k − c_i)
13:        Memo_ik = max{skip, count}
14:    return Memo_ik
```

Notar que este algoritmo difiere del código fuente proporcionado para la materia, ya que el pseudocódigo lo utilizamos para demostrar la idea del algoritmo de Programación Dinámica y no para mostrar explícitamente

el método de implementación. Cabe recalcar que ninguna de las diferencias entre el pseudocódigo y el código fuente alteran la complejidad del algoritmo Habiendo mencionado ésto, veamos que la complejidad de éste algoritmo *top-down* pertenece a $O(n \times M)$. El único caso que podemos notar que hay una mayor complejidad que $O(1)$ es al momento de hacer los llamados recursivos, siendo que el resto de las operaciones son comparaciones, asignaciones y lecturas sobre números y sobre la estructura de memoización, que podemos acceder en $O(1)$ gracias a que conocemos los índices donde estamos leyendo/escribiendo. Entonces, en los llamados recursivos nos damos cuenta de que estamos recorriendo la secuencia de locales 1 sola vez, porque nuestras iteraciones recursivas ocurren de manera salteada sobre los elementos de la secuencia hasta llegar al final, es decir, nosotros recorremos de una secuencia $L = l_1, l_2, l_3, l_4, ..., l_n$ por un lado los elementos $l_1, l_3, l_5, ...$ y por otro los elementos $l_2, l_4, l_6, ...$, gracias a lo mencionado previamente que denotábamos como una restricción del espacio de soluciones. Notemos que no importa si estos elementos recorridos efectivamente aportan o no al cálculo del beneficio máximo, sino que los recorremos de todas maneras 1 vez y evaluamos si son útiles o no. Y al acceder y asignar valores sobre la matriz, cuyo tamaño es $n \times M$, nos damos cuenta claramente de que en el peor caso la recorrida sobre ésta estructura pertenecería a la complejidad temporal de $O(n \times M)$, lo cuál sabemos que no necesariamente debe suceder en la práctica. Por lo tanto, podemos determinar que la complejidad temporal de peor caso del algoritmo de Programación Dinámica pertenece a $O(n \times M)$.

3. Experimentación

3.1. Introducción

En esta sección vamos a realizar una serie de experimentos con varios objetivos

- Medir complejidades temporales y tiempos de ejecucion en distintas secuencias de locales para cada método.
- Comparar la rapidez de los algoritmos entre sí a fin de demostrar como se comportan y ver cuales son “mejores” dependiendo de la entrada.
- En el caso de los algoritmos de backtracking, analizar métricas ejecutando el algoritmo sin podas, con una de las dos o con sendas.
- Comparación global entre los 3 algoritmos sobre las mismas entradas.

Para esto se correran una serie de tests; algunos con entradas completamente normales y otros con “edge cases” los cuales pondran a prueba a los algoritmos en casos poco convencionales. Todas estas pruebas seran realizadas en un entorno de trabajo equipado con una CPU AMD Ryzen 5 3600x clockeado a 3.8GHz con 6 nucleos y 12 hilos, 16GB de memoria RAM a 3000MHz y utilizando el lenguaje de programación C++.

Las familias de entradas o instancias que usaremos para experimentar buscaran recrear mejores casos, peores casos, casos típicos con variables aleatorias y en el caso del método de programación dinámica, crearemos una instancia dedicada y distinta al resto.

Observación: las siguientes instancias de experimentación fueron generadas manualmente por problemas de configuración de las laptops de Jupyter.

3.2. Terminología

Abreviaremos las configuraciones para los experimentos de la siguiente manera:

- **BF:** Algoritmo 1 de Fuerza Bruta de la sección 1.
- **BT:** Algoritmo 2 de Backtracking de la sección 2 sin ninguna poda activada.
- **BT-O:** Análogo a **BT** pero con la poda por optimalidad activada (sin factibilidad).
- **BF-F:** Análogo a **BT** pero con la poda por factibilidad activada (sin optimalidad).
- **BF-C:** Análogo a **BT** pero ambas podas activadas.
- **PD:** Algoritmo 3 de Programación Dinámica de la sección 3.

3.3. Experimento 1: Complejidad Fuerza Bruta

En esta sección analizaremos la complejidad del método de Fuerza Bruta. Dado que el método de Fuerza Bruta tiene una complejidad de $O(n \times 2^n)$, no plantearemos un mejor y peor caso. En cambio, utilizaremos instancias con distintos tamaños de entrada para analizar como aumenta el tiempo de ejecucuión en relación a ella. Tomando en cuenta que la función que describe la complejidad temporal del algoritmo es una función exponencial, esperamos que con instancias de entrada chicas tenga un crecimiento muy leve, pero a medida que se incrementa el tamaño se llegue a una tardanza muy elevada,tanto así que comienza a ser contraproducente utilizar éste algoritmo para resolver el problema, sabiendo que conocemos un algoritmo de menor complejidad temporal (Programación Dinámica). Sin embargo, cabe enfatizar que para instancias chicas, puede funcionar tan bien como cualquiera de los otros dos algoritmos que fueron descriptos a lo largo del informe.

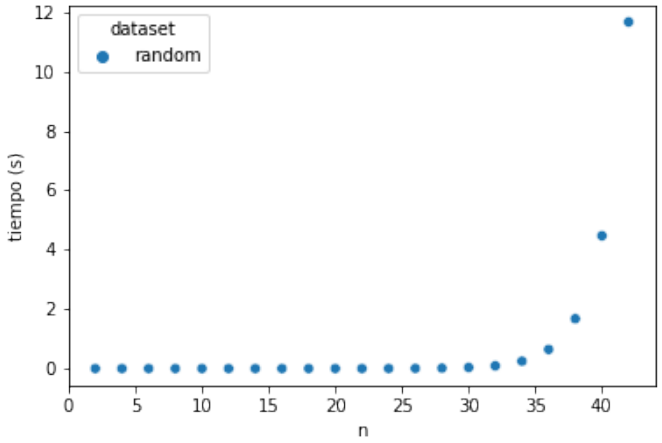

Figura 1: Duración en segundos del método de Fuerza Bruta con n de 2 a 42, step de 2

Para hacer una breve explicación y descripción del gráfico propuesto, vemos que llamamos a nuestras instancias de testeo *random* ya que probamos con secuencias con tamaños n y valores *aleatorios* para hacer el análisis de la manera menos sesgada posible, usando el eje y como el eje de tiempo (en segundos) para demostrar su progreso en el tiempo. De ésta manera podemos apreciar que sobre instancias de tamaño relativamente chico el gráfico refleja que el algoritmo funciona eficientemente, sin ningún pico extraño. Pero podemos ver también que ya acercándose a una instancia de testeo de $n = 40$ crece exponencialmente, de manera muy acelerada.

3.4. Experimento 2: Backtracking

Para la experimentación sobre el algoritmo de backtracking, creemos oportuno hacer un análisis de la efectividad de sus podas, por lo que primero repasamos los posibles mejores y peores casos para el algoritmo, y luego pasaremos a ver los datos reflejados en el gráfico propuesto. Recordamos que en el peor caso, el algoritmo de backtracking no hace uso eficiente de sus podas, por lo que no terminaría cortando ramas del árbol de recursión sobre el que trabaja, sino que generaría el conjunto de partes de los locales y elegiría al final de este proceso aquel con mayor beneficio. En cambio, el mejor caso ocurre cuando la instancia provista al algoritmo está estructurada de tal manera que permite usar eficientemente las podas, y recortar lo más posible las subinstancias no válidas, llegando más rápidamente a la solución correcta. En nuestro caso particular, utilizamos tanto podas de factibilidad como podas de optimalidad, que serán analizadas en el gráfico a continuación:

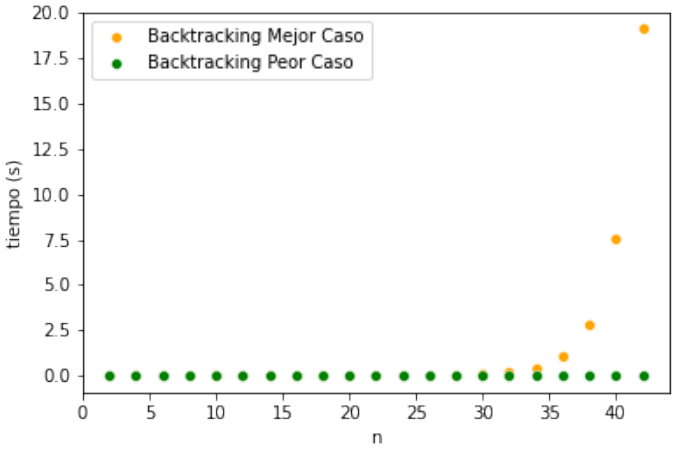

Figura 2: Duración en segundos del método de Backtracking en mejor y peor caso con n de 2 a 42

Podemos apreciar como *forzando* al algoritmo a pasar por el mejor caso y por el peor caso, hay una clara diferencia en crecimiento temporal a medida que va creciendo el tamaño de la entrada. Si bien el análisis de éstos casos es extremo (es decir, creamos situaciones extremadamente benéficas o perjudiciosas para la ejecución del algoritmo), es interesante notar la diferencia cuando las podas actúan y cuando no lo hacen. Quizás el gráfico podría nutrirse de más instancias con mucho más margen de crecimiento para sus tamaños, pero consideramos que de ésta manera continuaba siendo evidente la discrepancia entre complejidades que queríamos ver.

3.5. Experimento 3: Programación Dinámica

Dado que sabemos que la complejidad temporal teórica del peor caso del algoritmo de PD es exponencialmente mejor que la de los algoritmos de BF y BT por lo comentado en el análisis a lo largo del informe, creemos apropiado hacer un estudio del tiempo entre estos tres algoritmos, utilizando la misma instancia en ellos, para lograr hacer una comparación lo más objetiva y visual posible de la complejidad temporal en acción.

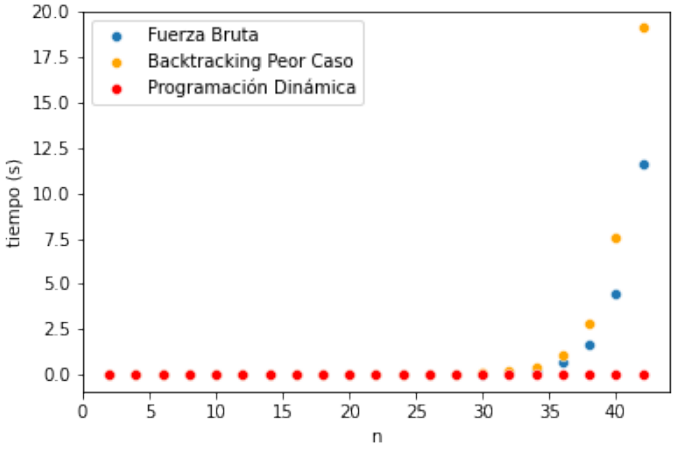

Figura 3: Duración en segundos del método de Fuerza Bruta con n de 2 a 42, step de 2

En el gráfico podemos apreciar cómo al analizar los peores casos de las 3 técnicas algorítmicas está más que claro la diferencia entre las implementaciones con complejidades temporales exponenciales y la del algoritmo de Programación Dinámica, que prácticamente parece no sufrir ningún crecimiento a comparación de BF y BT. El apoyo de la lógica extra y la estructura de memoización para implementar el algoritmo hacen visible la eficiencia superior de tal método algorítmico para la resolución de éste problema.

4. Conclusiones

En éste trabajo se presentaron 3 algoritmos que utilizaban 3 técnicas algorítmicas distintas para resolver el problema de Negocios por Medio / Distancia Social propuesto por la materia de Algoritmos y Estructuras de Datos III. Pudimos ver las diferencias en simplicidad de código y en eficiencia de procesamiento. Sin lugar a dudas el algoritmo más eficiente fue el de Programación Dinámica, pero lo destacamos como una aproximación mucho más compleja para la resolución del problema. Sin embargo, somos conscientes luego de la experimentación y análisis de los algoritmos que sería una muy buena opción para instancias que requieran mucha escalabilidad sobre el tamaño de entradas. De ésta manera, vemos que el costo de implementación se vió *remunerado* con una buena efectividad temporal de resolución, lo cuál es el caso opuesto al caso de fuerza bruta, donde es un algoritmo muy simple pero temporalmente costoso.
Quizás como punto de mayor estudio, podría continuar evaluándose posibles estructuras de datos para la estructura de memoización de PD.
De ésta manera, concluimos el análisis de los algoritmos para la resolución del problema de Negocios por Medio.