DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Técnicas Algorítmicas

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Alamino, Federico Julian	316/17	federicoalamino2@gmail.com
Giambastiani, Sebastian Matias	916/12	seba.giamba@gmail.com

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

Índice

1. Introducción

En este trabajo presentaremos una serie de algoritmos que brindan una solución al proyecto denominado NPM y los analizaremos a fin de estudiar algunas características tales como su complejidad temporal y rendimiento bajo distintos escenarios. Para ellos, ofreceremos su pseudocódigo, calcularemos su complejidad temporal teórica y realizaremos experimentos sobre sus implementaciones para corroborar de manera empírica algunas hipótesis planteadas.

1.1. El proyecto

NPM fue pensado bajo el contexto de la pandemia por COVID-19: intenta dar una alternativa para abrir los locales comerciales de una avenida que mayor beneficio generan para la economia, mientras se mantiene controlado el riesgo de expandir la enfermedad.
Formalmente podemos modelar el problema de la siguiente manera: dada una secuencia de n locales $L = [1, 2, ..., n]$, b_i el beneficio económico que aporta el local i, y c_i el riesgo de contagio que genera el local, con $i \in L$ y $b_i, c_i \in \mathbb{N}_{\geq 0}$, y dado M el límite de contagio tolerable, con $M \in \mathbb{N}_{\geq 0}$, una solución factible de NPM consiste en la suma de los beneficios de un conjunto de locales L' $\subseteq$ L que cumplen la siguientes condiciones:

1. $\sum_{i \in L'} C_i \leq M$
2. $i \in L' \implies i + 1 \notin L$

La solución buscada, considerada "óptima", será el mayor beneficio entre las soluciones factibles. En caso de no existir, el resultado será 0.

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

Si $L = [1, 2, 3, 4, 5]$, $b = [50, 25, 10, 20, 15]$, $c = [10, 10, 20, 30, 20]$ y $M = 40$ entonces existen varias soluciones factibles pero el mayor beneficio de todas las combinaciones posibles se obtiene abriendo los locales 1 y 4, por lo tanto la solución es 70 $(50 + 20)$.

Por otro lado, si tenemos $L = [1, 2, 3]$, $b = [10, 30, 50]$, $c = [20, 20, 15]$ y $M = 10$ entonces no existe ninguna solución factible, dado que el riesgo de contagio de cada local es mayor al límite tolerable. Por lo tanto en este caso no hay solución posible, y la respuesta es 0.

2. Algoritmos

2.1. Fuerza Bruta

2.1.1. Descripción

Los algoritmos de Fuerza Bruta permiten obtener la solución óptima a un problema analizando todas las soluciones posibles del mismo, incluso aquellas que no son factibles. Este es el caso del algoritmo propuesto, que analiza incluso las combinaciones de negocios que no cumplen la condición de contigüidad. Nuestro algoritmo además reconstruye la solución (aunque no la devuelve) utilizando un conjunto de índices para representar a los negocios. Esto último permite dividir al algoritmo en dos partes bien diferentes. Por un lado, el caso recursivo, que solo se encarga de realizar dos llamados para obtener los parámetros de la fúnción máximo: uno que incluye al índice actual, y otro que no. Por el otro, el caso base, que se encarga de recorrer los índices del conjunto y realizar tanto la suma total de los beneficios como la comprobación de las dos condiciones requeridas por el problema. Esta manera simple de diseñar el algoritmo facilita la comprensión del mismo, reduce

los errores y nos permite utilizarlo como referencia para comprobar el correcto funcionamiento del resto de las implementaciones. El caso recursivo es reutilizable, ya que construye un conjunto de partes genérico, mientras que la lógica propia del problema se encuentra en el caso base.

Algorithm 1 Algoritmo de Fuerza Bruta para NPM

```
1: function FB(i, indices)
2:     if i ≥ n then
3:     return sumarBeneficio(indices) else                    ▷ Caso Base
4:     return máx{FB(i + 1, indices ∪ {i}),FB(i + 1, indices))}   ▷ Caso Recursivo
```

Algorithm 2 Función auxiliar *sumarBeneficio*

```
1: function sumarBeneficio(indices)
2:     beneficio = 0
3:     contagio = 0
4:     while i < indices.length do
5:         beneficio = beneficio + indices[i].beneficio
6:         contagio = contagio + indices[i].contagio
7:         if i ≥ 1 ∧ indices[i] ≤ indices[i − 1] + 1 then return 0   ▷ Chequea Contigüidad
8:         if contagio > M then return 0                          ▷ Chequea Contagio
9:         i = i + 1
10:    return beneficio
```

2.1.2. Complejidad

Cada llamado a la función FB realiza a su vez dos llamados recursivos y aumenta el índice i en una unidad hasta llegar a n. Tenemos entonces 2^n llamados **recursivos**. Cada llamado realiza una operación en $O(1)$: llamar recursivamente y calcular el máximo, mientras que los 2^n casos base llaman a la función sumarBeneficio que realiza a lo sumo n comparaciones. Por esta razón, el algoritmo tiene complejidad $O(n * 2^n)$.

2.2. Backtracking

Los algoritmos de Backtracking se basan en árboles recursivos, en los cuales los ramas son las desiciones que se van tomando. Los árboles pueden ser podados, ya sea por factbilidad (que cumpla la condición establecida) o por optimalidad (se descartan las soluciones peores). Esta técnica algorítmica es idéntica a la utilizada para fuerza bruta, y en particular Backtracking sin podas tiene la misma complejidad en peor caso.

Las podas planteadas para la solución al problema NPM han sido las siguientes:

Poda por factibilidad: Aquí el foco estuvo centrado en una de las restricciones del problema: el límite de contagio máximo que pueden tener la totalidad de negocios simultaneamente abiertos (M).

Cuando el algoritmo está recorriendo un nodo y el contagio del local actual más el contagio de los locales previamente abiertos supera M, entonces ese local y todos los que le siguen no cumplen con la condición, y por lo tanto se poda toda esa rama.

Poda por optimalidad: La poda por optimalidad está modelada en base al beneficio máximo obtenido actualmente.

Sea $ACUM$ el beneficio calculado actualmente por una determinada rama, si BT está procesando una instancia en la cual el beneficio de L[i] sumado a todo el beneficio restante de la secuencia L es menor al beneficio encontrado en alguna rama, entonces se elimina automaticamente la rama

procesada, ya que el beneficio obtenido será menor que el actual, por lo tanto no es una solución óptima al problema.

Si benef(L[i]) es el beneficio acumulado hasta el i-esimo local, y n=‖L‖

$$benef(L[i]) + \sum_{j=i}^{n} benef(L[j]) < ACUM$$

2.3. pseudo-codigo

Siendo n la cantidad de negocios, M el límite de contagio permitido, max el maximo beneficio hasta el negocio i-esimo, y hayMas una función que calcula en O(n) la optimalidad de la solución presentada, el siguiente algoritmo descripto en pseudo-codigo representa la solución planteada para el ejercicio:

Algorithm 3 Algoritmo de Backtracking para npm

```
1:  function BT(i, benef, contagio, NoContiguo, UltAgregado)
2:      if i = n then
3:          if contagio<=M && max<benef && NoContiguo then
4:              max = benef
5:              return benef
6:          else
7:              return 0;
8:      if contagio > M then
9:          return 0;
10:     if !hayMas(benef,i) then                          ▷ hayMas ∈ O(n)
11:         return 0;
12:     return máx{BT(i+1, benef+benef(i), contagio+contagio(i), NoContiguo&&abs(ultAgregado−
        i)≥ 2,i), BT(i+1, benef, contagio,NoContiguo,UltAgregado) }.
```

Algorithm 4 Algoritmo auxiliar para BT

```
1:  function HAYMAS(benef,i)
2:      acum = benef
3:      while i≥0 do
4:          acum += negocio[i].beneficio
5:          i - -
6:      return acum ≥ max                          ▷ max = maximo beneficio actual
```

2.3.1. Complejidad

La complejidad del algoritmo en el peor caso es $O(n * 2^n)$. Esto sucede, en el caso de no podar ninguna rama, por lo cual termina siendo un algoritmo de fuerza bruta, el cual dicha complejidad temporal resulta ser la misma. Las operaciones realizadas por el algoritmo en su mayoría son en tiempo constante, es decir $O(1)$, pero cuando se quiere calcular la optimalidad de una solución, se utiliza una función que en tiempo lineal calcula lo planteado anteriormente. Por otro lado, la poda por factibilidad, asegura que se recorrerá a lo sumo 2^n veces el árbol, dependiendo del contagio máximo permitido. Así como para la complejidad en el peor caso no cambia mucho mas que una constante con dicha poda, para el mejor caso hace el algoritmo lineal, esta solución se debe a cuando el contagio permitido es menor al contagio en cada uno de los n-locales, el algoritmo poda cada una de las diferentes ramas.

2.4. Programación Dinámica

2.4.1. Descripción

El algoritmo de programación dinámica propuesto implementa un árbol de backtracking similar al de los algoritmos anteriores. Sin embargo, realizamos una modificación para evitar generar combinaciones innecesarias: una poda por factibilidad implícita en la forma de generar las soluciones de las ramas de la izquierda. En la figura 1 se muestra un árbol de backtracking convencional de 4 niveles. En cada llamado recursivo se decide si se suma o no el beneficio del local actual (comenzando por el local 1) y luego en ambos llamados recursivos se incrementa en 1 el índice del próximo local a calcular.

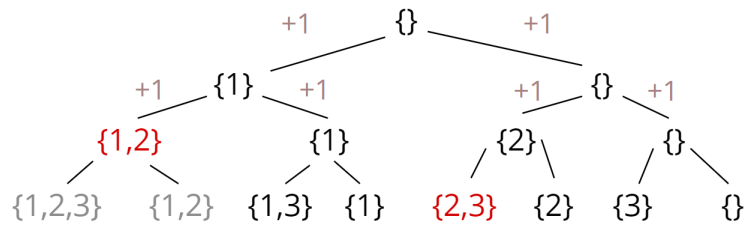


Figura 1: Árbol de BT de 4 niveles con incremento +1 en ambas ramas.

Se puede ver en rojo los nodos que no cumplen la condición de contigüidad y en gris sus hijos, que por herencia tampoco cumplirán esta condición.

Si incrementamos en 2 el índice del próximo local a calcular para los hijos de la izquierda del árbol conseguimos evitar los nodos innecesarios y obtener un árbol como el que sigue:

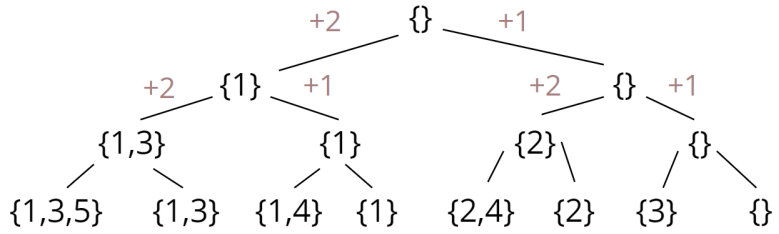


Figura 2: Árbol de BT de 4 niveles con incremento +2 en rama izquierda.

Gracias a esta poda implícita, cada ejecución recursiva de la función solo debe chequear que la suma del riesgo de contagio acumulado con la del local a calcular no supere el límite permitido y por lo tanto depende únicamente de sus parámetros de entrada: índice del local y contagio acumulado (y por supuesto de la secuencia de locales y el límite de contagio definidos por la instancia que está corriendo). Esto permite memoizar los resultados de los llamados que utilizan los mismos parámetros en una matriz de $(n + 1) * M$. De esta forma, al llamar a PD solo se calcula el valor del nodo actual si el mismo no se calculó previamente. La cantidad de llamados no memoizados posibles da origen a la complejidad teórica del algoritmo: $O(n * M)$. En la siguiente imágen podemos ver el arbol de llamados recursivos. Se puede notar que a partir del tercer nivel los llamados comienzan a repetirse (se indican con un mismo color). En el caso en que el contagio

acumulado (el segundo parámetro) coincida con el de un llamado anterior al mismo índice, el algoritmo recortará esa rama y no seguirá ejecutando:

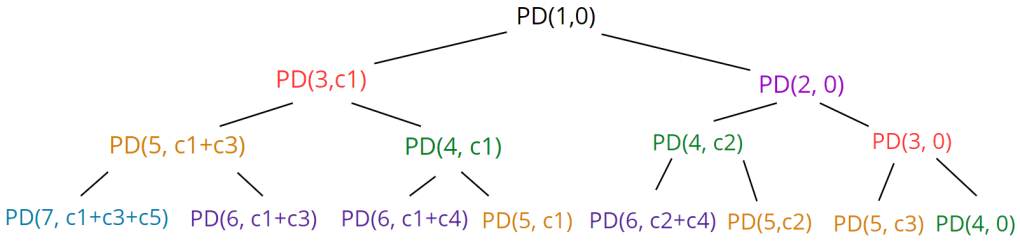


Figura 3: Árbol de llamadas a función PD, de 4 niveles con incremento +2 en rama izquierda.

Algorithm 5 Algoritmo de Programación Dinámica para NPM

```
1: function PD(i, c)                          ▷ i: índice del local, c: contagio acumulado
2:     if memoization[i][c] == VACIO then
3:         if i >= n ó c + c_i > M then
4:             memoization[i][c] = 0                                          ▷ Caso Base
5:         else
6:             memoization[i][c] = max(PD(i + 2, c + c_i) + b_i, PD(i + 1, c))   ▷ Caso Recursivo
7:     return memoization[i][c]
```

3. Experimentación

El código de los algoritmos fue escrito en c++, mientras que los experimentos y gráficos se realizaron en Python. La experimentación completa se encuentra disponible para replicar en los archivos fuente del proyecto. La máquina utilizada corría Ubuntu 18.04 con un procesador Intel Core i7-7500U.

3.1. Experimento 1: Complejidad de Fuerza Bruta

3.1.1. Hipótesis

Como mencionamos anteriormente los algoritmos de fuerza bruta garantizan resolver el problema para cualquier instancia utilizando una lógica simple y fácil de implementar. Como contraparte, su complejidad temporal suele ser elevada y se vuelven ineficientes para tamaños de entrada grandes. En particular, nuestro algoritmo utiliza un arbol de backtracking, pero por el hecho de que requiere evaluar todas las posibles combinaciones de resultados no es posible hacer podas y por lo tanto el tiempo que tarda en correr el algoritmo depende exclusivamente del tamaño de la entrada (en este caso la cantidad de locales). La hipótesis planteada para este experimento es que los tiempos medidos se corresponderán con la complejidad teórica calculada: $O(n * 2^n)$, y que el algoritmo se comportará de manera idéntica para cualquier familia de instancias que se le provea. Solo veremos variaciones en función del tamaño del conjunto de datos provisto, ignorando así las características que hacen de una instancia un mejor o peor caso.

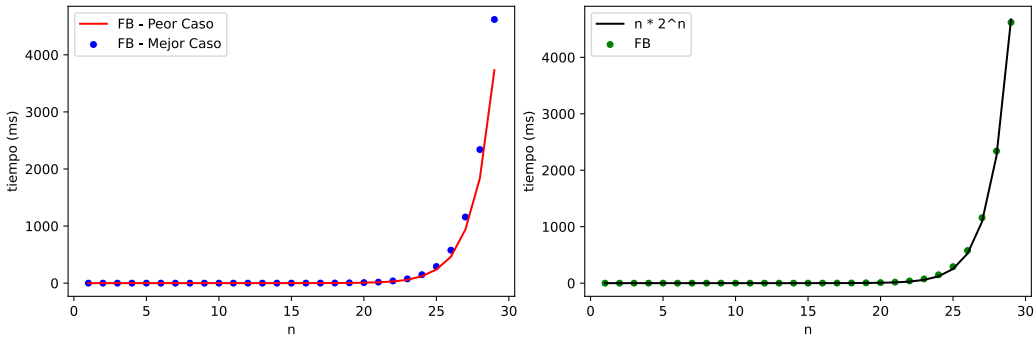
3.1.2. Diseño

Tomamos dos conjuntos de instancias distintos. En ambos casos realizamos mediciones variando la cantidad de locales, comenzando por un único local hasta llegar a un total de 31. El primer conjunto se asegura que la solución involucre un único local convenientemente ubicado al final de

la instancia. De esa forma modelamos un peor caso para un árbol de backtracking. El segundo conjunto busca exactamente lo contrario: ubicamos la solución como un único local al comienzo de la instancia asegurándonos que el resto de los locales no sean solución (cada uno tiene un contagio mayor al límite máximo).

3.1.3. Resultados

Los resultados confirman la hipótesis. Los gráficos muestran que los tiempos medidos para ambos conjuntos de instancias son idénticos y además los mismos se corresponden con la curva $n * 2^n$ corroborando así la complejidad teórica calculada anteriormente.



(a) Complejidad peor caso vs mejor caso

(b) Tiempo medido vs esperado

Figura 4: Tiempos de ejecución del algoritmo FB

3.2. Experimento 2: Complejidad peor caso Backtracking

3.2.1. Hipótesis

Backtracking es eficiente a la hora de podar ramas y «*quitar*» soluciones no factibles, sin ello tendria la misma complejidad que un algoritmo planteado con fuerza bruta; por eso en esta experimentación se tratara de corroborar que sucede si no se podan esas ramas. Por lo tanto, la premisa para dicho experimento se basa en analizar empiricamente la complejidad del peor caso del algoritmo de backtracking planteado. La hipotesis manejada refiere a que la complejidad deberia ser igual al calculo teorico mencionado anteriormente, O(n * 2^n), siendo n la cantidad de locales.

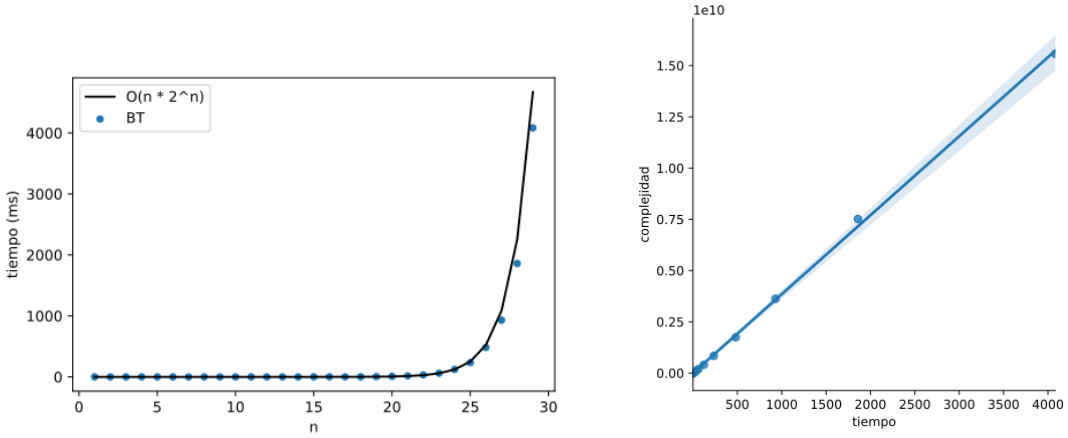
3.2.2. Diseño

El peor caso fue modelado a partir de un item clave, que nunca se pode, es decir que nunca se corte el algoritmo con las podas de factibilidad u optimalidad, y que rearme y recorra todo el arbol y las funciones auxilares tantas veces como negocios haya. Para lograr esto la instancia se modelo de la siguiente forma:

- El ultimo elemento esta en la solución final.
- La suma del contagio de todos los locales es menor al contagio maximo permitido
- Que la suma los beneficios de todos locales excepto el ultimo sean menor al beneficio del ultimo.

Esos tres items fuerzan al algoritmo a generar todo el arbol de backtracking, es decir recorrer todas las posibles soluciones en $O(2^n)$ y por cada ejecución, la funcion auxiliar de optimalidad se ejecuta en $O(n)$. Lo que me daria como resultado $O(n * 2^n)$

3.2.3. Resultados



(a) Complejidad

(b) Coef. correlación

Figura 5: Complejidad Backtracking

Como se puede ver en los gráficos, se corrobora de manera empírica la complejidad en el peor caso con la calculada, siendo esta $O(n * 2^n)$

3.3. Experimento 3: Comparación de podas en Backtracking

3.3.1. Hipótesis

En esta experimentación queremos ver el comportamiento del algoritmo de backtracking al correr diferentes familias de instancias con distintas podas.

Hipotesis 1: La eficiencia del algoritmo al aplicarle las podas depende de la distribución de los datos. No hay una poda mejor que otra. Hipotesis 2: Ninguna poda es mejor que la otra en todos los casos.

3.3.2. Diseño

La idea de este experimento es observar el comportamiento del algoritmo para un modelo de datos más realista. Para ello consideramos dos posibles distribuciones de beneficios para los locales y dos posibles distribuciones de niveles de contagio.

Para el beneficio consideramos, por un lado, que los locales están sobre una avenida en la cual existe una intersección de calles que es centro comercial. Los locales que están cerca de esa intersección son más concurridos y por lo tanto generan un beneficio económico mayor. Decimos que en este caso aplicamos un beneficio INCREMENTAL dado que el beneficio se incrementa a medida que nos acercamos al centro comercial. Por otro lado, si no existe tal centro comercial, decimos que el beneficio es CONSTANTE, ya que todos los locales toman un valor aleatorio dentro de un rango no variable.
Para el contagio de los locales consideramos una distribución con contagio ALTO y otra con contagio BAJO. En la primera, cada local tiene un nivel de contagio que está entre el 33 % y el 50 % del M (máximo tolerable). En la segunda cada local toma un nivel de entre 1 y el 25 % del M.

3.3.3. Resultados

Los gráficos muestran que nuestras hipótesis son ciertas y nos dan detalles del funcionamiento de las podas.
Cuando el nivel de contagio es BAJO respecto a M, la poda por factibilidad no es muy eficiente. Esto se debe a que el algoritmo necesita recorrer muchos locales para obtener un valor de contagio acumulado que supere a M y podar así la rama actual. El experimento muestra que para estos conjuntos de datos es más útil aplicar la poda por optimalidad. Sin embargo no podemos garantizar que esta relación se mantenga para todos los casos.
Por otro lado, cuando el nivel de contagio es ALTO se puede ver que la poda por factibilidad es mejor opción que la poda por optimalidad. Se puede observar también, a partir de los tiempos medidos, cómo esta última poda mejora los tiempos del experimento para el conjunto de datos con beneficio INCREMENTAL. Esto se debe a que el algoritmo genera el árbol de backtracking recorriendo primero los locales que están al final de la lista. En estos casos, el algoritmo encuentra rápidamente una solución que involucra los primeros locales recorridos, y cuyo beneficio acumulado supera a la suma de los beneficios del resto de los locales que aún restan por recorrer. De esa forma, se poda la rama actual, evitando así procesar muchas soluciones.

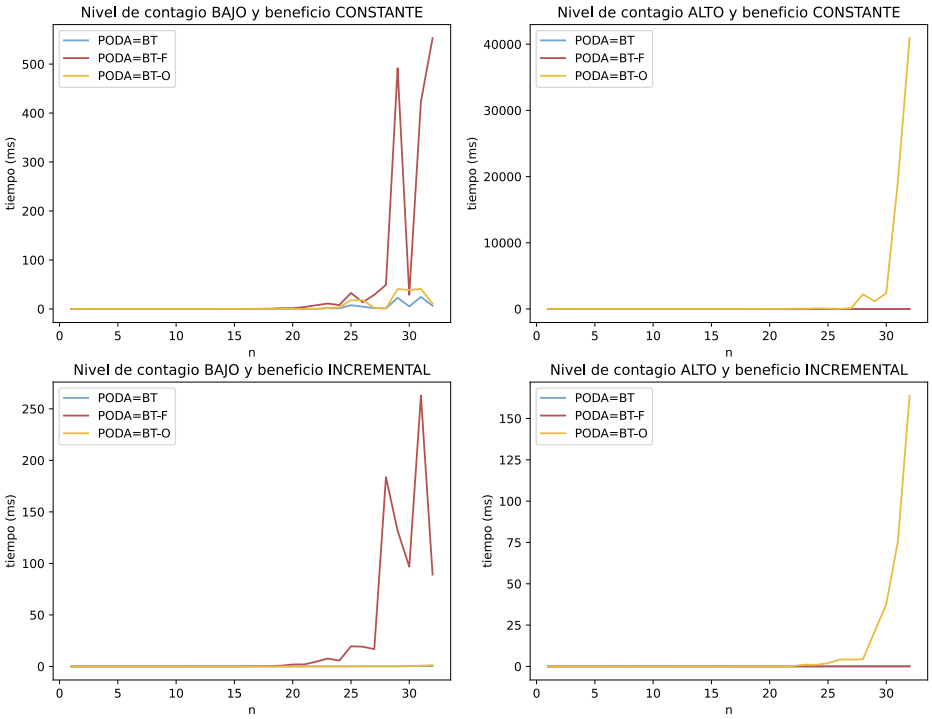


Figura 6: Comparación de podas

3.4. Experimento 4: Complejidad en Programación Dinamica

3.4.1. Hipótesis

Queremos corroborar la hipótesis de que el algoritmo de programación dinámica tiene complejidad teórica $O(n * M)$.

3.4.2. Diseño

Debido a que la complejidad involucra dos variables, analizamos cada una por separado, fijando primero una y variando la otra (y viceversa). Para realizar el experimento creamos 4 conjuntos de instancias, una para cada variable fijada. Esto nos permite tener una idea de qué ocurriría si se movieran ambas al mismo tiempo. En todos los casos, los locales toman un nivel de contagio pseudo-aleatorio (obtenido mediante la función randomint de python 3) entre el valor 1 y el valor M del conjunto.

3.4.3. Resultados

Los resultados muestran un incremento en los tiempos a medida que aumenta el valor de la variable del eje x, en ambos casos mientras se fija la otra. Al compararlos con la función lineal de la variable medida se puede deducir que tanto n como M influyen aumentando linealmente los tiempos del algoritmo.

También se observan fluctuaciones en las mediciones. Esto se debe principalmente a que los valores de contagio por local son randomizados y no siguen un patron determinado.

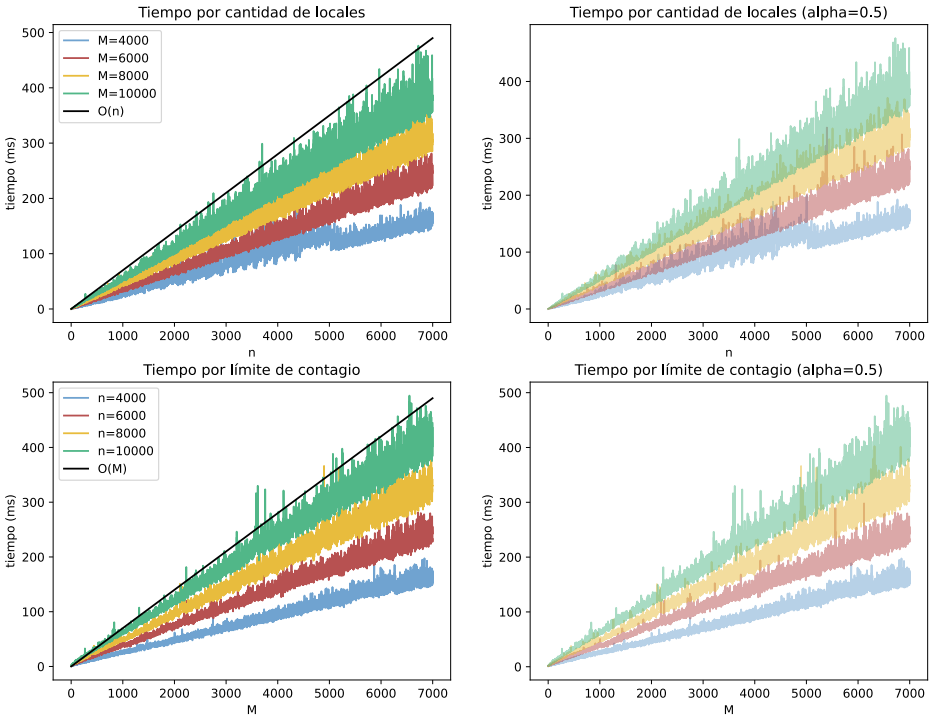


Figura 7: Complejidad Programación dinamica

4. Conclusiones

Los algoritmos propuestos resuelven el problema original planteado (NPM). Sin embargo, el cálculo de sus complejidades y su posterior corroboración mediante experimentación nos enseña que las decisiones tomadas a la hora de elegir la técnica algorítimica son cruciales para que el problema se pueda implementar de manera eficiente. El algoritmo FB es siempre la peor solución en términos de performance y apenas puede implementarse para instancias pequeñas de cerca de 30 locales (en tiempo razonable). Por otro lado, al incluir algunas podas al árbol puede mejorarse sustancialmente la performance, según lo visto en el algoritmo BT. Sin embargo es necesario conocer que algunos casos esas podas no funcionan y las instancias toman tiempo exponencial. El algoritmo de PD permite correr las instancias mayores en tiempo razonable y además ofrece la posibilidad de controlar linealmente el crecimiento mientras se pueda asegurar que el conjunto de datos mantiene fija la cantidad de locales o el contagio máximo permitido. Si bien no lo analizamos en este trabajo, para esta última técnica se debe disponer suficiente memoria para guardar los tiempos calculados.