Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Algoritmos y Estructuras de Datos III

Distanciamiento social

Francisco Peretti & Laureano Muñiz

fmperetti@hotmail.com, lau2000m@hotmail.com

Índice general

1. INTRODUCCIÓN

A raíz de la pandemia que estamos atravesando se generaron muchos inconvenientes en la sociedad, siendo uno de los más importantes el cierre de tiendas en las avenidas principales de la ciudad. Con el objetivo de reabrir los comercios para generar una reactivación económica, se creo un proyecto llamado **negocio por medio**, en el cual se estudiaron los comercios de las avenidas principales y se les asoció a cada uno el valor de beneficio económico y el valor de contagio que provoca su apertura.

El objetivo de este trabajo practico es implementar y estudiar algoritmos para poder llevar a cabo este proyecto. Se nos provee un valor de límite de contagio LC y una secuencia de n tiendas $T = [t_0, t_1, ..., t_{n-1}]$, donde cada tienda t_i cuenta con un índice de beneficio y de contagio asociado, b_{ti}, $c_{ti} \in \mathbb{N}_0$. Nuestros algoritmos deben identificar cual es el beneficio total que genera la subsecuencia de tiendas $T' \subseteq T$ óptima. Una subsecuencia óptima es, en este caso, aquella que maximiza el beneficio total cumpliendo con dos restricciones:

- No pueden existir elementos contiguos de la secuencia original. Es decir,

$$noTieneContiguos(T') \Leftrightarrow (\forall i \in \mathbb{N}_0)(i \geq 0 \wedge i \leq n-1 \wedge_L t_i \in T' \Rightarrow_L t_{i+1} \notin T')$$

- El límite de contagio debe acotar la suma total de los índices de contagio de los elementos de la subsecuencia seleccionada. Definimos que el contagio de T' es permitido si y solo si está acotado superiormente por LC:

$$contagioPermitido(T') \Leftrightarrow \sum_{t \in T'} c_{ti} \leq LC$$

Definimos que T' es **factible** si y solo sí cumple las primeras 2 condiciones, decimos que:

$$esFactible(T') \Leftrightarrow noTieneContiguos(T') \wedge contagioPermitido(T') \tag{1.1}$$

Nos interesa, entonces, calcular el beneficio total de la subsecuencia que cumple las restricciones y maximiza el beneficio total. En otras palabras, queremos seleccionar una subsecuencia **factible** de T, que satisfaga que la suma de los beneficios de sus tiendas sea mayor a la de cualquier otra subsecuencia **factible** de T.
Sea T' una subsecuencia de T tal que esFactible(T') decimos que:

$$T' \; maximiza \; el \; beneficio \Leftrightarrow \forall T'' \subseteq T \wedge_L esFactible(T'') \Rightarrow_L \sum_{t \in T''} b_{ti} \leq \sum_{t \in T'} b_{ti}$$

Podemos recrear un ejemplo con 5 locales. Si tenemos el contagio limite $LC = 50$, los beneficios $b = [30, 40, 30, 50, 10]$ y los contagios $c = [30, 10, 15, 20, 30]$, el beneficio máximo a obtener es 90, usando la subsecuencia $T' = [t1, t3]$. Advertimos que no necesariamente seleccionar una mayor cantidad de tiendas o sumar una mayor cantidad de contagio implica maximizar el beneficio total.

2. DESARROLLO

Como nos enfrentamos a un problema de optimización, la primer idea que nos surge es que debemos obtener y comparar todas las posibles soluciones para obtener aquella que sea la mejor de todas. En esta ocasión, realizaremos esto implementando un algoritmo de Fuerza Bruta. Pensamos al conjunto de soluciones posibles como un árbol binario, donde el primer nodo padre es la solución vacía, la rama izquierda de cada nodo denota agregar la siguiente tienda a la solución parcial y la rama derecha denota no agregarla. Notar que el árbol binario tiene 2^n hojas que representan todas las soluciones posibles.

Rápidamente nos damos cuenta que comparar la totalidad de posibilidades, aunque sirve para obtener una solución al problema, no es la mejor de nuestras opciones. Esto se debe a que la cantidad de posibilidades aumenta exponencialmente en función de la cantidad de elementos de la secuencia inicial de tiendas. Por lo tanto, decidimos buscar la manera de acotar la cantidad de posibilidades analizadas. Para esto, nos proponemos preguntarnos características sobre la solución parcial a medida que la construimos para saber si realmente puede ser una candidata a resolver el problema. Una manera de acotar es preguntar si la solución parcial es **factible**, dando lugar a la implementación de un algoritmo de Backtracking con podas de factibilidad. Esta idea para acotar consiste de que si solución parcial que estamos construyendo no es **factible**, entonces ninguna solución que se genere a partir de ella lo va a ser tampoco. De esta manera, se va a descartar esa solución parcial y no se van a investigar las soluciones de las cuales dicha solución parcial es prefijo, disminuyendo así la cantidad de soluciones a comparar.

Siguiendo con una mentalidad similar, entonces nos surge la pregunta, ¿No habrá alguna forma de acotar las posibilidades que sea mejor o que complemente a la de factibilidad? Buscamos entonces generar alguna condición que de lugar a un algoritmo de Backtracking con podas de optimalidad. Al analizar el problema, nos damos cuenta que hay un valor que siempre acota al beneficio máximo total, este es la suma de todos los beneficios de todas las tiendas juntas. Pese a que notamos que dicho valor se va a conseguir únicamente si el subconjunto que contuviera a todas las tiendas fuera factible, esta lógica de acotar el beneficio con la suma de las tiendas es muy interesante. Esta idea la podemos extrapolar a cualquier nodo del árbol de posibilidades. Siempre vamos a poder acotar el máximo de cualquier solución parcial con la suma del beneficio de todas las tiendas que queden por contemplar y el beneficio parcial acumulado de esa solución parcial. Si nosotros a medida que encontramos soluciones posibles llevamos un registro de cual fue la solución mas óptima hasta al momento, luego vamos a poder comparar la máxima suma potencial de una solución parcial contra el actual candidato a solución óptima. Si la máxima suma potencial de una parcial, que desconocemos siquiera si es factible, no supera al candidato a máximo actual, entonces no tiene sentido seguir explorando la rama de esa solución. De esta manera, podemos implementar así una poda por optimalidad y reducir la cantidad de soluciones a comparar.

A partir de estas ideas, nos proponemos investigar sobre el rendimiento de las distintas estrategias para obtener la solución al problema. ¿Será que las podas por optimalidad por si solas son mas eficientes para resolver el problema? ¿O será que lo son las de factibilidad?¿Es lo más eficiente complementar ambas?

Por último, también intentamos pensar alguna estrategia distinta para obtener el mejor beneficio que no recayera tanto en pensar a la estructura del conjunto de soluciones como un árbol binario. En base a esto, planteamos entonces pensar que el problema podía ser resuelto usando los resultados de subproblemas más pequeños. Por ello, intentamos definir una función de forma recursiva de manera tal que los resultados de la función se pudieran inferir de resultados anteriores. Al realizar esta función, identificamos que habría varios llamados recursivos con los mismos parámetros que se repetirían innecesariamente. Por lo cual, aplicando una técnica de *memoización* decidimos guardar los resultados ya calculados en una matriz, implementando un algoritmo de Programación Dinámica.

2.1. Fuerza Bruta (FB)

Nuestro algoritmo de fuerza bruta se basa en buscar todas las soluciones posibles al problema sin poner demasiada atención al momento de obtenerlas. Pensamos al conjunto de soluciones como un árbol binario, lo que nos lleva a implementar un algoritmo recursivo para facilitar el trabajo sobre este tipo de estructura. Lo que caracteriza al algoritmo de FB es que todos los llamados recursivos van a terminar en el caso base. El algoritmo se basa en una función que recibe parámetros a modo de flag, para recopilar información sobre la posible solución a medida que se construye, para luego tomar una decisión sobre la solución posible cuando se termine de construir, es decir, cuando llegue hasta el caso base. Nuestra función recibe 5 parámetros, 2 que funcionan a modo de flags, 2 que funcionan a modo de acumulador y otro que funciona a modo de iterador.

El iterador, representa la tienda actual, Cada vez que hacemos un llamado recursivo descartamos una tienda del conjunto original, de modo que hacemos 2 llamados recursivos por cada tienda, uno considerando agregar la tienda a la solución parcial y otro en el que no la agregamos. En cada llamada se consulta al iterador contra el numero de tiendas totales (`línea 3 del pseudocódigo`), si nuestro iterador llegó al numero de tiendas totales entonces sabemos que ya exploró todas las tiendas para esa solución parcial por lo que se ejecuta el caso base y devuelve.

Los flags son `noHayContiguos` y `último` que se van a ayudar entre si para definir si la solución parcial que estamos construyendo se formó con tiendas contiguas o no, esto se maneja de la siguiente manera: En cada llamado a la función, esta se va a llamar recursivamente 2 veces, un llamado va a elegir agregar la tienda actual a la solución parcial y otra va a elegir no hacerlo (`línea 10 del pseudocódigo`), en este momento al hacer los llamados recursivos el valor del flag `último` para los siguientes llamados se va a definir dependiendo de si eligen agregar la tienda actual o no. Así mismo el flag `noHayContiguos` va a decidir su valor para los siguientes llamados en base al valor de `último` y se quiere agregar la tienda actual, además del valor actual de `noHayContiguos`. En caso de que se quieran agregar contiguos, el flag `noHayContiguos` va a tomar el valor `false`. Esto es muy importante ya que al llegar al caso base, nosotros vamos a decidir ignorar una solución si `noHayContiguos` está en `false`, más allá de que el contagio esté en el rango permitido y el beneficio sea mayor al máximo actual.

Los acumuladores, `ben_parcial` y `cont_parcial`, están para tener un registro del beneficio y el contagio total que genera la solución. A medida que la función se ejecuta y avanza en los llamados recursivos suceden dos casos. En el caso donde se agrega la tienda a la solución parcial, vamos a sumar el beneficio y el contagio a los acumuladores `ben_parcial` y `cont_parcial` respectivamente. En el caso de que no se agregue vamos a conservar el valor actual ya que representa la suma del contagio y beneficio de todas las tiendas que se agregaron hasta el momento. En especial, al llegar al caso base estos acumuladores nos van a ser útiles para definir 2 cosas, si el contagio parcial supera al límite de contagio establecido entonces vamos a descartar la solución (ya que no es **factible**), y en el caso de que eso no suceda, y además el flag de `noHayContiguos` sea `true` entonces vamos a devolver al beneficio como resultado. (`líneas 4 a 8 del pseudocódigo`)

Podemos suponer que este algoritmo de FB implementado nos lleva a la solución del problema ya que, se evalúan la totalidad de posibilidades y con ayuda de los flags y de los acumuladores se tiene información suficiente para definir si es una solución realmente **factible**.

La complejidad es $\mathcal{O}(2^n)$, ya que en cada llamado recursivo se van a ejecutar siempre 2 otros llamados, el caso base es $iterador = n$, y tiene complejidad temporal constante. Por lo que concluyendo, $2^n * \mathcal{O}(1) = \mathcal{O}(2^n)$

Algorithm 1 Algoritmo de Fuerza Bruta

```
1: cantTiendas ← T.size()
2: function FB(t, nhc, u, c, b)    ▷ tiendaActual, noHayContiguos, último, cont_parcial, ben_parcial
3:     if t = cantTiendas then
4:         if nhc = True ∧ c ≤ LC then
5:             return b
6:         else
7:             return 0
8:         end if
9:     else
10:        return max{FB(t+1, ¬u∧nhc, True, c+T[t].c, b+T[t].b), FB(t+1, nhc, False, c, b)}
   ▷ Obtener el máximo de los hijos
11:    end if
12: end function
```

2.2. Backtracking

Los algoritmos de Backtracking desarollan una idea similar a los de Fuerza Bruta pero realizan ciertas consideraciones especiales para evitar procesar casos innecesarios. De la misma manera que en Fuerza Bruta, se considera al conjunto de soluciones como un árbol binario. Lo particular es que en BT se analiza nodo a nodo si la solución parcial puede ser solución del problema. En el caso de que una solución parcial no cuente con ciertas condiciones específicas, se puede evitar explorar las soluciones que se extiendan de esta. El hecho de evitar considerar ramas del árbol de soluciones se denomina *poda*, y se la suele clasificar en *factibilidad* y *optimalidad* de acuerdo a las condiciones que determinan si se debe analizar o no el resto de la rama.

2.2.1. Poda de factibilidad

La poda de factibilidad se produce cuando al estar construyendo una solución parcial se pudo recopilar suficiente información sobre la misma para definir que no es **factible**, por ello nuestro algoritmo trabaja con un flag y un acumulador para definir en cada momento si la solución parcial es o no **factible**. En este caso, el acumulador registra el contagio acumulado y el flag es el que determina si la tienda anterior inmediata se agregó a la solución parcial o no. En todos los llamados a la función, que no sean el caso base, se hacen consultas sobre el flag y el acumulador para evaluar si se sigue extendiendo la solución parcial (`línea 8 del pseudocódigo`). Se producirá una poda cuando el flag indique que la tienda anterior inmediata pertenece a la solución parcial ya que no tiene sentido en este contexto seguir investigando la rama que extendería a esta solución agregando la actual, ya que estas dos serían contiguas y la factibilidad de la solución se perdería. También se podaría cuando el acumulador que indica el contagio superase al limite de contagio establecido. Podemos asumir que vamos a obtener la solución al problema ya que, con la ayuda del flag y el acumulador, nos aseguramos de investigar todas las soluciones factibles. Luego, al llegar al caso base tenemos información necesaria para determinar el beneficio acumulado de cada una de las soluciones factibles, tomamos el máximo de todos esos beneficios.

Algorithm 2 Algoritmo de Backtracking con Poda de Factibilidad

```
1:  cantTiendas ← T.size()
2:  function BTF(t, u, c, b)                    ▷ tienda, último, cont_parcial, ben_parcial
3:      if t = cantTiendas then
4:          return b
5:      else
6:          hijoIzquierdo ← 0
7:          hijoDerecho ← 0
8:          if ¬u ∧ c + T[t].c ≤ LC then
9:              hijoIzquierdo ← BTF(t + 1, True, c + T[t].c, b + T[t].b)
10:         end if
11:         hijoDerecho ← BTF(t + 1, False, c, b)
12:         return max{hijoIzquierdo, hijoDerecho}        ▷ Obtener el máximo de los hijos
13:     end if
14: end function
```

2.2.2. Poda de optimabilidad

Por otro lado, la poda de optimalidad se produce cuando a medida que se está construyendo una solución posible, se recopiló suficiente información para determinar que la misma no proporciona un resultado mejor al de otra solución ya explorada. En este caso, nuestro algoritmo cuenta con un acumulador encargado de guardar el beneficio potencial que pueden sumar las tiendas aún no consideradas en la solución y un registro que guarda el candidato a máximo beneficio de las soluciones ya exploradas. Cuando se analiza un nodo, se verifica si es que el beneficio parcial acumulado sumado al beneficio potencial del acumulador pueden superar al actual candidato a máximo(`línea 15 del pseudocódigo`). En caso de que no lo supere, se puede podar toda la rama consiguiente a ese nodo, puesto que ninguna solución de esa rama podría llegar a generar un beneficio tal como para superar al actual máximo. Notamos que el candidato a máximo se actualiza únicamente cuando se explora hasta una hoja del árbol, se verifica que la solución es factible y el beneficio total que provee es mayor al del candidato a máximo.(`línea 8 y 9 del pseudocódigo`) Podemos determinar que este algoritmo es correcto puesto que compara todas las soluciones posibles, pero evita explorar ramas innecesarias que no puedan superar el candidato a máximo beneficio. Nuevamente, este algoritmo es $\mathcal{O}(2^n)$. En el peor caso, donde todos los locales superan el limite de contagio, la poda de optimalidad nunca puede descartar ramas porque nunca se actualiza el candidato a máximo. De esta manera, el algoritmo recorre todos los nodos del árbol binario verificando diferentes condiciones en $\mathcal{O}(1)$.

Algorithm 3 Algoritmo de Backtracking con Poda de Optimalidad

```
1: maximo ← 0
2: cantTiendas ← T.size()
3: sumaRestanteInicial ← Σ_{t∈T} b_t ▷ El algoritmo debe ser llamada por primera vez con este valor
4:
5: function BTO(t, nhc, c, b, u, sr)      ▷ tienda, noHayContiguos, cont_parcial, ben_parcial, último,
6:                                                                                  ▷ sumaRestante
7:     if t = cantTiendas then
8:         if nhc = True ∧ c ≤ LC ∧ maximo < b then
9:             maximo ← b
10:        end if
11:        return maximo
12:    else
13:        hijoIzquierdo ← 0
14:        hijoDerecho ← 0
15:        if b + sr > maximo then
16:            hijoIzquierdo ← BTO(t + 1, ¬u ∧ nhc, c + T[t].c, b + T[t].b, True, sr − T[t].b)
17:            if b + sr > maximo then        ▷ Reevaluar la guarda por si maximo cambio su valor
18:                hijoDerecho ← BTO(t + 1, nhc, c, b, False, sr − T[t].b)
19:            end if
20:        end if
21:        return max{hijoIzquierdo, hijoDerecho}                ▷ Obtener el máximo de los hijos
22:    end if
23: end function
```

2.2.3. Combinación de podas

Creamos un algoritmo que combina las dos podas para evitar explorar la mayor cantidad de soluciones innecesarias. En la teoría, este algoritmo aún pertenece a $\mathcal{O}(2^n)$ puesto que las podas no son lo suficientemente fuertes como para poder bajar la complejidad asintótica. Sin embargo, esperamos que en la práctica tenga una *performance* mejor que los algoritmos que solo realizan un tipo de podas.

Algorithm 4 Algoritmo de Backtracking con Podas Combinadas

```
1: maximo ← 0
2: cantTiendas ← T.size()
3: sumaRestanteInicial ← ∑_{t∈T} b_t ▷ El algoritmo debe ser llamado por primera vez con este valor
4:
5: function BT(t, c, b, u, sr)          ▷ tienda, cont_parcial, ben_parcial, último, sumaRestante
6:     if t = cantTiendas then
7:         if maximo < b then
8:             maximo ← b
9:         end if
10:        return maximo
11:    else
12:        hijoIzquierdo ← 0
13:        hijoDerecho ← 0
14:        if b + sr > maximo then
15:            if ¬u ∧ c + T[t].c ≤ LC then
16:                hijoIzquierdo ← BT(t + 1, c + T[t].c, b + T[t].b, True, sr − T[t].b)
17:            end if
18:            if b + sr > maximo then      ▷ Reevaluar la guarda por si maximo cambio su valor
19:                hijoDerecho ← BT(t + 1, c, b, False, sr − T[t].b)
20:            end if
21:            return max{hijoIzquierdo, hijoDerecho}      ▷ Obtener el máximo de los hijos
22:        end if
23:        return 0
24:    end if
25: end function
```

2.3. Programación Dinámica

Desarrollamos un algoritmo que cambia el enfoque del problema a resolver. En este caso, dejamos de pensar el problema asociado a un árbol de soluciones, y comenzamos a pensar que este podía ser dividido en subproblemas más pequeños. Diseñamos una función recursiva que nos permite plantear el algoritmo. Dada la lista de n tiendas T, con sus beneficios y contagios asociados b_{t_i}, $c_{t_i} \in \mathbb{N}_0$ y el límite de contagio LC, planteamos la función $f(i, c)$:

$$f(i,c) = \begin{cases} 0 & \text{si } i \geq n \vee c < 0 \\ f(i+1,c) & \text{si } c_{t_i} > c \\ max\{b_{t_i} + f(i+2, c-c_{t_i}), f(i+1,c)\} & \text{en otro caso} \end{cases}$$

Sin ir más lejos, $f(i, c)$ indica cual es el máximo beneficio que se puede conseguir utilizando alguna combinación factible de tiendas en $T' = [t_i, t_{i+1}, ..., t_n]$ teniendo un contagio máximo de c. Analizamos lo que realiza la función en cada caso:

Si $i \geq n \vee c < 0$, estamos en un caso donde o bien T' es una secuencia vacía de tiendas o el contagio permitido no permitiría agregar más tiendas. Es por esto que en este caso el beneficio a retornar es 0. Si $c_{t_i} < c$, estamos en un caso donde el contagio asociado de la tienda t_i es mayor al contagio permitido, por lo que no se puede considerar como una tienda para agregar.

En cualquier otro caso, vamos a querer obtener el máximo entre $(b_{t_i} + f(i+2, c-c_{t_i}))$ agregar el beneficio parcial y continuar extendiendo la solución ignorando la tienda contigua, y $(f(i+1, c))$ no agregar la tienda actual y extender la solución con la tienda contigua. Notar que este método también tiene en cuenta la factibilidad, al considerar el caso de sumar la tienda, ya que directamente salta la tienda siguiente ya que sabe por defecto que esta no se va a poder sumar. Notamos que esto no cambiaría ni el contagio restante ni el beneficio parcial ya que si o si la tienda siguiente inmediata no se va a sumar por cuestiones de factibilidad.

Utilizamos la ya mencionada ténica de *memoización* para rellenar una tabla de $n * LC$ posiciones con los resultados de esta función. Puesto que en un peor caso se debe llenar la mayoría de la tabla, acotamos superiormente y afirmamos que este algoritmo pertenece a $\mathcal{O}(LC * n)$

Algorithm 5 Algoritmo de Programación dinámica

```
1: M_j k ← ⊥ for j ∈ [0, n], k ∈ [0, LC].
2: function DP(i, c)
3:     if i > n ∨ i < 0 then return 0
4:     end if                                          ▷ Si i está fuera de rango
5:     if c > LC ∨ c < 0 then return 0
6:     end if                                          ▷ Si c está fuera de rango
7:     if M[i][c] = ⊥ then                  ▷ Si este problema aún no ha sido resuelto
8:         if i = n then                              ▷ Si el subconjunto considera 0 locales
9:             M[i][c] ← 0
10:        end if
11:        if T[i].c > c then        ▷ Si el contagio de la tienda actual es mayor al contagio restante
12:            M[i][c] ← DP(i + 1, c, M)
13:        else
14:            M[i][c] ← max{T[i].b + DP(i + 2, c − T[i].c, M), DP(i + 1, c, M)}
15:                    ▷ Obtener el máximo entre las opciones de agregar la tienda y no agregarla
16:        end if
17:    end if
18:    return M[i][c]
19: end function
```

3. EXPERIMENTACIÓN

3.1. Metodología

Todos los experimentos fueron ejecutados en una computadora con las siguientes cualidades: Procesador: Ryzen 5 3600 a una frecuencia de 4.2 Ghz con 6 núcleos de CPU y un total de 12 hilos de ejecución, RAM: 16GB a 3 Ghz

Con el fin de analizar bajo qué situaciones los algoritmos son más eficientes, se diseñaron las siguientes familias de instancias:

- **Caso real**: A partir de la idea de que todas las tiendas tienen un índice de contagio parecido en la realidad, se creó una familia de instancias donde todas las tiendas tienen un contagio similar. Definimos arbitrariamente un LC = 1000, y definimos un valor $P = \frac{2*LC}{n}$, siendo n la cantidad de locales. Luego, establecimos que cada tienda t va a tener un contagio asociado que sigue una distribución uniforme en el rango $[0{,}95\ P; 1{,}05\ P]$. Es evidente que existe una gran cantidad de tiendas que genera poco beneficio como las peluquerías, los kioskos, los bazares, y una cantidad pequeña de tiendas que genera gran beneficio como las concesionarias de autos o las tiendas de venta de inmuebles. Por esta razón es que decidimos que cada tienda tendrá un beneficio asociado que sigue una distribución χ^2 cuadrada con 3 grados de libertad, multiplicada por 10.
- **Caso random**: Elegimos realizar una familia de instancias aleatorias. En esta familia, el contagio límite se elige de forma aleatoria en el rango $[1, n]$ donde n es la cantidad de tiendas de la instancia. Los contagios asociados se eligen de forma aleatoria en el rango $[1, i * 4]$, donde i es el número de tienda. Por otro lado, los beneficios asociados se eligen en el rango $[1, LC]$.
- **Peor caso BT**: Diseñamos el peor caso para el algoritmo que combina podas. Para poder evitar la poda de factibilidad lo máximo posible, debemos asegurar que los contagios acumulados por las posibles soluciones tiendan a ser menores que el límite de contagio. Para poder evitar la poda de optimalidad lo máximo posible, debemos asegurar que los candidatos a máximo beneficio no sean mayores a una solución a una solución aun no vista. Establecemos entonces que dado una lista de n tiendas, la enésima tienda debe aportar un valor significativamente más alto que el resto de las tiendas y su valor asociado de contagio debe estar por encima del valor límite de contagio. Elegimos que todas las tiendas salvo la última tengan 0 de contagio y beneficio asociado.
- **Mejor caso BT**: Diseñamos el mejor caso estableciendo que ningún subconjunto sea factible por los niveles de contagio y que el beneficio asociado a cada tienda sea 0. De esta manera, la podas de factibilidad y optimalidad deberían determinar que no existe solución en poco tiempo.
- **Caso orden**: Construimos instancias de manera similar al caso random pero decidimos ordenar las tiendas de acuerdo a su beneficio asociado. Creamos una familia de instancias ordenadas de manera creciente y otra familia de instancias ordenadas de manera decreciente.

3.2. Experimentos

3.2.1. Experimento 1: Complejidad de Fuerza Bruta

En este experimento nos concentramos en analizar el rendimiento de nuestro algoritmo de Fuerza Bruta. Vamos a poner a prueba el algoritmo contra las 5 familias distintas de instancias. Tenemos 2 hipótesis en relación a este experimento:

- Dada una cantidad de tiendas el rendimiento va a ser el mismo para cualquier tipo de instancia.
- La complejidad teórica va a ser muy similar a la complejidad empírica($\mathcal{O}(2^n)$).

A continuación se muestran los gráficos de la experimentación de Fuerza Bruta:

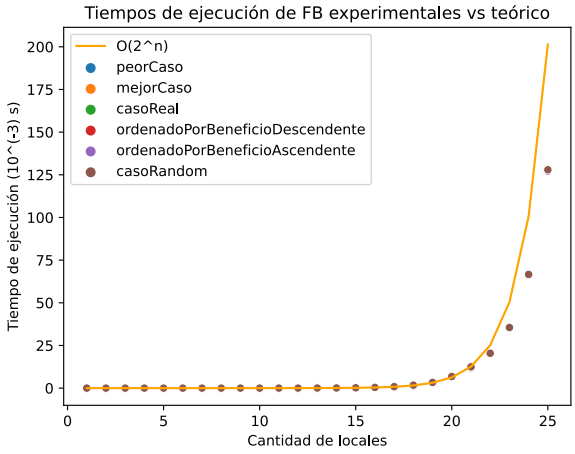

Fig. 3.1: Gráfico de tiempos de ejecución de Fuerza Bruta comparados con el tiempo esperado de ejecución

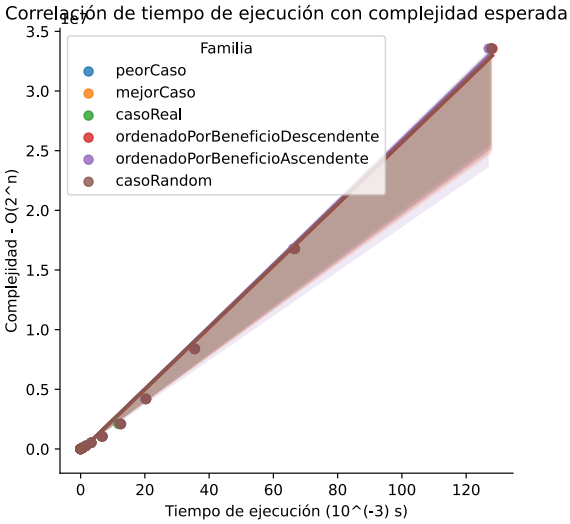

Fig. 3.2: Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada

Observando la Figura 3.1 advertimos claramente que para cualquier instancia de la misma cantidad de tiendas, el tiempo que tarda el algoritmo en calcular la respuesta es igual. Además, observamos que los puntos del gráfico, que corresponden al tiempo de ejecución, realizan una curva de naturaleza similar a la curva de complejidad esperada. Existe una pequeña diferencia entre la curva teórica y la curva experimental que se debe a una constante multiplicativa que omitimos. En ese sentido, podemos corroborar observando la Figura 3.2 que la correlación con la con la complejidad esperada es casi perfecta puesto que todos los tiempos de ejecución se encuentran sobre la recta de regresión lineal obtenida. Además, obtuvimos que el índice de correlación de Pearson es de 0.9987368856765877.

3.2.2. Experimento 2: Peor y mejor caso de Backtracking

En este experimento nos vamos a centrar en analizar la *performance* de nuestros algoritmos de Backtracking en su peor y mejor caso.

Al pensar en el comportamiento en el peor caso de las distintas implementaciones de los algoritmos de Backtracking nos surgen intuitivamente las siguientes hipótesis:

- En el peor caso la implementación que combina ambas podas será menos performante que las implementaciones de solo factibilidad u optimalidad, ya que se ejecutan bastantes mas guardas por llamado recursivo en el algoritmo de Backtracking de podas combinadas.
- No debería haber mucha diferencia de rendimiento entre Backtracking de poda de factbilidad y Backtracking de poda de optimalidad en el peor caso.
- En el peor caso, los algoritmos de Backtracking tienen una complejidad de $\mathcal{O}(2^n)$

A continuación se muestran los gráficos de la experimentación de peor caso:

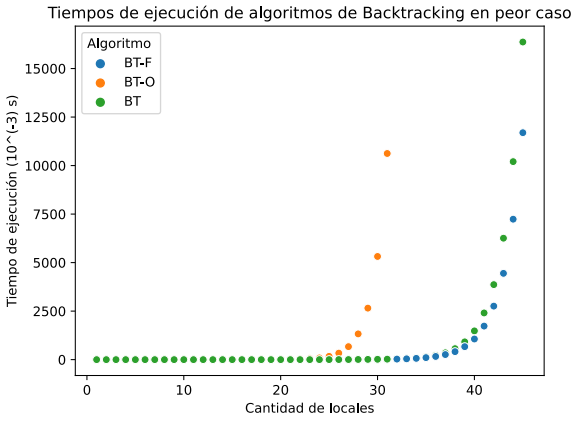

Fig. 3.3: Gráfico de tiempos de ejecución de algoritmos de Backtracking en el peor caso

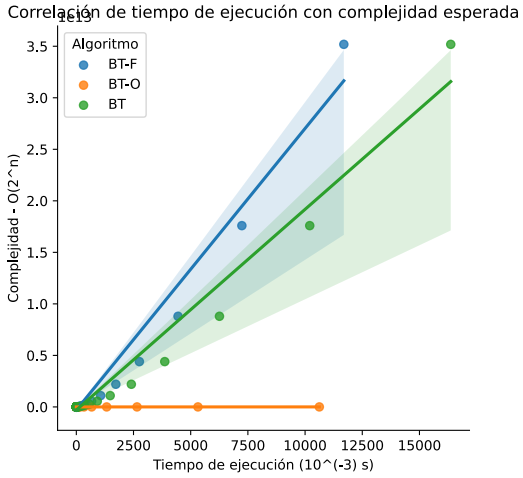

Fig. 3.4: Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada

En la Figura 3.3 advertimos que nuestras dos primeras hipótesis fueron refutadas empíricamente. Observando el gráfico nos damos cuenta que la curva del algoritmo de Backtracking con poda de optimalidad tiene una constante mucho más alta que los otros dos algoritmos. También reconocimos que el comportamiento asintótico de Backtracking con las dos podas y Backtracking con poda única de factibilidad son muy similares. Haciendo un análisis más profundo determinamos que esto se debe a que los algoritmos que tienen poda de factibilidad evitan una gran cantidad de nodos porque evitan prematuramente recorrer todos los nodos que tienen tiendas contiguas. Por otro lado, podemos determinar que en el peor caso, los algoritmos de Backtracking tienen una complejidad de $\mathcal{O}(2^n)$ por lo visto en las curvas de la Figura 3.3 y por la correlación graficada en Figura 3.4. Los índices de Pearson obtenidos para BT-F, BT-O y BT son 0.9853789321817054, 0.9999994561369887 y 0.9851153432102973 respectivamente.

Por otro lado, al pensar en el comportamiento en el mejor caso de las distintas implementaciones de los algoritmos de Backtracking nos surgen intuitivamente estas hipótesis:

- En el mejor caso la implementación que combina ambas podas y la que solo realiza poda por factibilidad serán similares y mejores que la que realiza poda de factibilidad.
- En el mejor caso, nuestros algoritmos de Backtracking tienen una complejidad de $\Omega(n)$ multiplicadas por constantes diferentes.

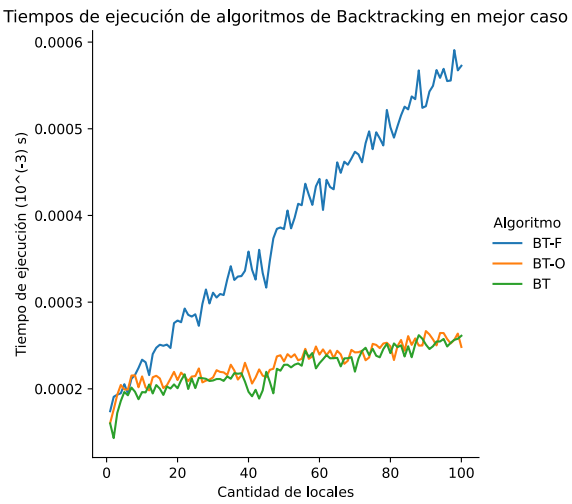

Fig. 3.5: Gráfico de tiempos de ejecución de algoritmos de Backtracking en el mejor caso

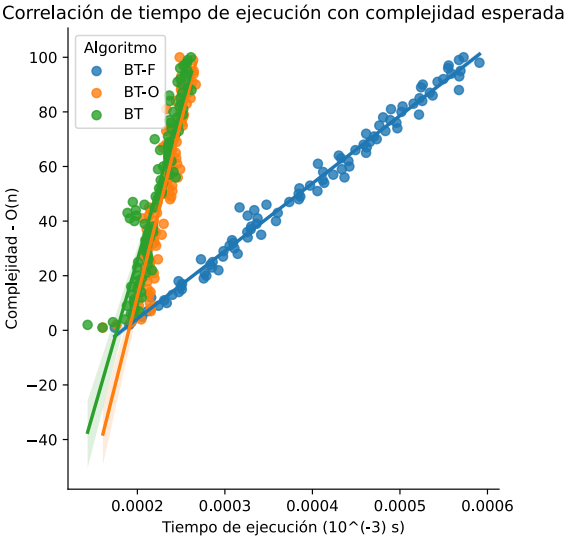

Fig. 3.6: Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada en el mejor caso

En este caso, nuestras hipótesis efectivamente se cumplen. Observamos en la Figura 3.5 que los dos algoritmos con poda de optimalidad presentan tiempos muy similares. Lo que sucede en estos algoritmos es que rápidamente podan las dos primeras ramas del árbol puesto que en la raíz del árbol comprueban que ninguna solución aportará una solución mejor que 0. El algoritmo que realiza poda de factibilidad por su parte debe recorrer n nodos del árbol realizando varias operaciones hasta determinar que no hay una posible solución. En la Figura 3.6 evidenciamos que los tres algoritmos pertenecen a ωn. Los índices de Pearson obtenidos para BT-F, BT-O y BT son 0.9942163500538644, 0.9184424593623591 y 0.9142367782203648, respectivamente.

3.2.3. Experimento 3: Orden en Backtracking

En este experimento vamos a comparar las distintas implementaciones de Backtracking sobre el tipo de instancia a la que definimos como Caso orden. En pos de que los resultados sean representativos generamos 10 instancias distintas de acuerdo a la cantidad de locales.
Nuestras hipótesis para exte experimento son:

- El orden de las tiendas afecta al rendimiento de BT podas combinadas. Más precisamente, un orden decreciente con respecto al beneficio va afectarle negativamente y el orden creciente va a afectarle de manera positiva haciendo que se produzcan mas podas

Graficamos los tiempos de ejecución de Backtracking con dos podas para distintas instancias random con diferentes ordenes para los beneficios asociados:

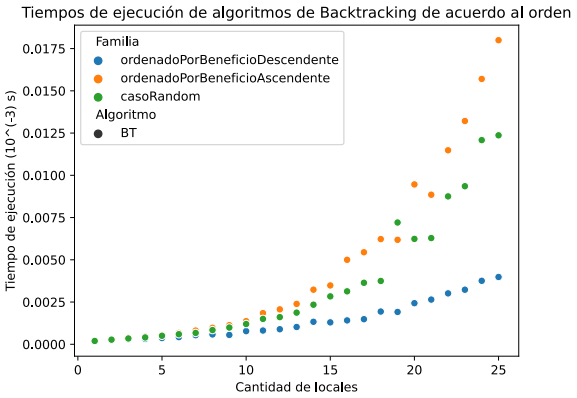

Fig. 3.7: Gráfico de tiempos de ejecución de Backtracking con ambas podas de acuerdo al orden

Observando la Figura 3.7 podemos ver que nuestra hipótesis se cumple, se puede ver que comparando el rendimiento de ambas instacias contra la instancia random, que tiene un comportamiento promedio, la instancia random se encuentra en el medio de las dos, acotando a una superiormente y a otra inferiormente. También se ve que el beneficio Descendente además está más alejado del caso random por lo que presenta un rendimiento bastante más favorable que el promedio, lo cual se ajusta a nuestra hipotesis inicial.

3.2.4. Experimento 4: Programación dinámica vs BT

En este experimento vamos a centrarnos en la *performance* de nuestra implementación con Programación Dinámica. En primer lugar, utilizando la familia de instancias de Caso real, corroboraremos cual es la complejidad de este algoritmo en función de los niveles de contagio y cantidad de locales. Elegimos realizarlo con este tipo de instancias ya que sostenemos que este es el algoritmo que se utiliza en la cotidianidad y debíamos probarlo con una instancia acorde a lo que se asemeja a la realidad. Nuevamente, en pos de que los resultados sean representativos generamos 10 instancias distintas de acuerdo a la cantidad de locales. Generamos un heatmap en función del contagio límite y la cantidad de locales para corroborar que la complejidad corresponde a $\mathcal{O}(LC * n)$:

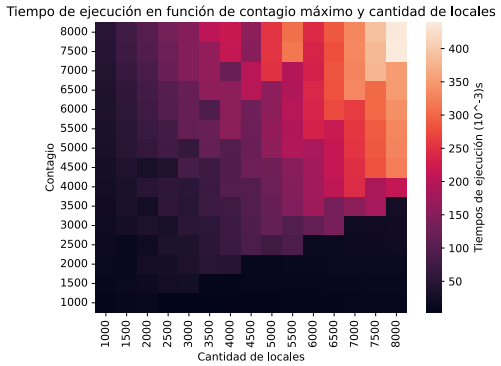

Fig. 3.8: Heatmap generado a partir de niveles de contagio y cantidad de locales

En la Figura 3.8 podemos observar que los cambios de complejidad no se dan igual por fila que como por columna. Esto si cambia en la diagonal del mapa, lo que quiere decir que los tiempos de ejecución están ligados a las distintas parejas n y LC. Advertimos que cuando uno de los dos parámetros esta fijo los cambios no son drásticos. Es por esto que en el heatmap los colores claros, que representan mayores tiempos, estan en la esquina superior derecha. Esto de alguna manera ratifica que el orden del algoritmo es $\mathcal{O}(LC * n)$.

Por otro lado, nos dedicaremos a comparar la *performance* entre los algoritmos de Backtracking y Programación Dinámica. Es importante mencionar que ningún algoritmo domina al otro en términos de complejidad. Es decir, no es cierto que $\mathcal{O}(2^n) \subseteq \mathcal{O}(LC * n)$ ni tampoco que $\mathcal{O}$(LC*n) $\subseteq \mathcal{O}(2^n)$. Teniendo esto en cuenta hipotetizamos que:

- Si calculamos para un rango pequeño de contagios, en determinada cantidad de tiendas en adelante, BT va a acotar superiormente a DP.

A continuación presentamos el gráfico obtenido al comparar BT vs DP para instancias de casoReal:

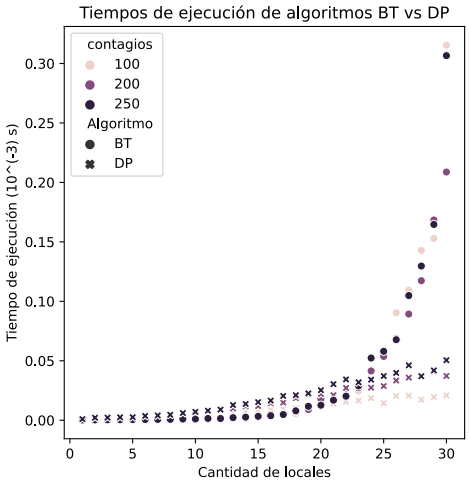

Fig. 3.9: Gráfico de tiempos de ejecución de algoritmos de BT vs DP

En la Figura 3.9 observamos que nuestra hipótesis se cumple. Además, evidenciamos que cuando $n \sim 23$, los tiempos de ejecución de BT comienzan a superar a los tiempos de DP en los contagios elegidos. Reflexionamos entonces que si el límite de contagio es pequeño, este es el mejor método para resolver este proyecto en particular. Sin embargo, sabemos que para un nivel de contagio arbitrariamente alto, deberíamos entonces considerar resolver el problema con BT.

4. CONCLUSIONES

Al proponernos investigar sobre 3 distintas técnicas algorítmicas para resolver un problema, partimos con la idea de encontrar la técnica definitiva, que resulte ser mejor que las otras en todo sentido. Pero mediante experimentos, que comparan la performance de todos los algoritmos, y analizando un seleccionado de tipos de instancia que deducimos podrían ser interesantes, pudimos reconocer que no hubo técnica que sea mejor en todo aspecto, sino que para distintas variaciones y parámetros de las instancias, conviene inclinarse por implementar un algoritmo u otro. Esto nos resulta sumamente interesante ya que hace énfasis en la idea de que además de tener que preocuparnos por la complejidad teórica asintótica de nuestro algoritmo, tenemos que considerar el contexto en el que se va a usar, para tomar una decisión inteligente sobre que técnica usar al implementar el algoritmo.

En conclusión este TP nos ayudó a entender mas a fondo las desventajas y ventajas de las distintas técnicas algorítmicas individualmente, y además a poder ver un contraste entre todas ellas, y así comprender, a grandes rasgos, cual implementar en cada situación.