DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Métodos de Programación

Algoritmos y Estructura de Datos III

Segundo Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Lucas Iván Kruger	799/19	Lucaskruger10@gmail.com
Sebastián Cantini Budden	576/19	sebascantini@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

En este informe implementaremos distintas técnicas de programación con el fin de analizar diferencias entre ellas. Para esto implementaremos un algoritmo por cada técnica que solucione el siguiente problema: Ante la aparición de nuevas variantes de la enfermedad que acecha el mundo, el gobierno quiere evitar una nueva cuarentena y para eso, se logro calificar cada local con un numero que indica el beneficio que produce a la economía y su nivel de riesgo, el problema a resolver es dada una avenida, cual es la mejor combinación de locales que generen el mejor beneficio sin que se supere un cierto nivel de riesgo, con la condición adicional de que no pueden haber dos locales consecutivos abiertos. Implementaremos, la solución al problema utilizando tres métodos: fuerza bruta, backtracking y programación dinámica, con el fin de poder encontrar las distintas ventajas que cada uno ofrece.

2. Implementación

Para simplificar la implementación y el manejo de la información de la avenida, optamos por generar una estructura para almacenar todos estos datos para un fácil acceso y simplificar la comprensión. Esta estructura a la cual nos referiremos como **info**, tiene los siguientes observadores:

- info.limite: capacidad limite de contagios.
- info.cantidad: cantidad de tiendas en la avenida.
- info.beneficios: arreglo de los beneficios de cada tienda individual ordenada geográficamente (como estarían situadas en la avenida).
- info.contagios: arreglo de nivel de riesgo que aporta cada tienda, nuevamente ordenada como info.beneficios.

Esta estructura la utilizaremos para acceder a la información en todos los algoritmos, que veremos a continuación. Esto agrega una complejidad de $O(n)$ con n siendo la cantidad de tiendas, ya que creamos dos arreglos de esa longitud. Sin embargo, no agrega a la complejidad temporal ya que esta implementada con arreglos, los cuales tienen complejidad de acceso $O(1)$.

2.1. Fuerza Bruta:

Nuestro primer algoritmo a analizar, es fuerza bruta. Este recorre todo el espacio de posibles soluciones (aunque sean invalidas, por ende el nombre) y calcula el beneficio y nivel de contagio, luego al final decide si el camino es valido. La implementación es simple ya que no hay que tomar nada en cuenta. Veamos la siguiente implementación:

Algorithm 1 bruteForce*(const info_t &info, int i, bool valido, bool consecutivo, int riesgo, int beneficio)*

1: **if** $i \geq info.cantidad$ **then**
2: **if** $(valido \neq true)||(riesgo > info.limite)$ **then**
3: $beneficio \leftarrow 0$
4: **end if**
5: **return** $beneficio$
6: **end if**
7: $b1 \leftarrow bruteForce(info, i+1, valido, false, riesgo, beneficio)$
8: $b2 \leftarrow bruteForce(info, i+1, valido\&\&!consecutivo, true, riesgo + info.contagios[i], beneficio + info.beneficios[i])$
9: **return** $max\{b1, b2\}$

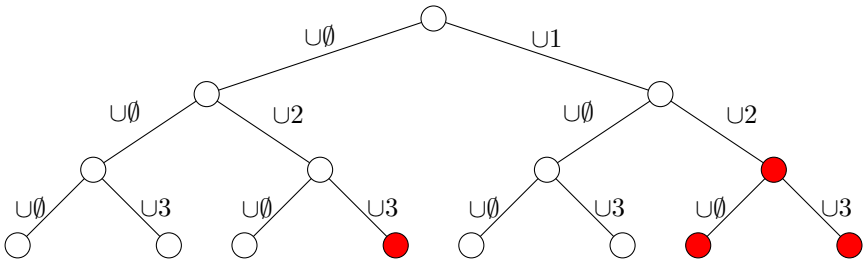

Figura 1: Ejemplo de ejecución del algoritmo de fuerza bruta donde usamos el primer llamado recursivo para ir bajar hacia la izquierda de un nodo y el segundo llamado para bajar hacia la derecha del mismo. Los nodos rojos son considerados caminos inválidos por concatenación consecutiva de tiendas.

Este pseudocódigo se divide en dos partes, de la linea (1) hasta la linea (6) y de ese punto en adelante. En la primera parte tenemos el caso base donde se corrobora que hayamos llegado al final de la avenida, antes de devolver el valor obtenido en ese camino, corrobora que sea valido (no hayan dos locales consecutivos) y que tal combinación no supere el limite de contagio elegido, devuelve cero si alguno de los dos casos mencionados anteriormente ocurre. La segunda parte del algoritmo es el llamado recursivo que permite recorrer la avenida. En la linea (7) se asume que la tienda i no se considera en la secuencia de tiendas a abrir, mientras en el segundo llamado recursivo si, se puede ver como en este segundo llamado, tomamos en cuenta los contagios y beneficios de la tienda i durante el llamado.

Este algoritmo tiene complejidad de $O(1)$ espacial ya que toda la información necesaria se pasa por parámetro y no requerimos estructuras adicionales y en cuanto a complejidad temporal, cada instancia se llama recursivamente dos veces, hasta que se llegue al final del conjunto de tiendas, por ende si tenemos n tiendas, tendremos una complejidad de $O(2^n)$.

Ahora veamos que efectivamente recorre todo el espacio de soluciones. Para esto consideraremos toda combinación de tienda como posible solución y luego determinaremos si la solución es valida y óptima o no. Nuestro algoritmo tiene dos llamados recursivos donde en el primero no aceptamos a la tienda actual como parte del conjunto que forma la posible solución y el segundo llamado si la acepta. Tal como vemos en la imagen de ejemplo de ejecución, este método logra todos los casos desde no aceptar ninguna tienda a aceptar todas.

A plena vista se puede ver la debilidad de esta técnica, este método recorre todas las posibles combinaciones, incluyendo las que son imposibles. Si vemos en la imagen de ejemplo de ejecución, podemos ver que hay tres caminos inválidos, esto es por que esa posible solución contiene dos tiendas consecutivas. En el problema a solucionar sabemos que no pueden haber dos tiendas consecutivas, sin embargo, este algoritmo también se fija los caminos con tiendas consecutivas costando tiempo adicional. Veamos, también, que hay un nodo que invalida todos los nodos inferiores, esto es por que la concatenación se hizo al principio de la cadena, invalidando todo el subárbol de soluciones.

2.2. Backtracking:

Backtracking intenta optimizar la fuerza bruta con distintas podas. Llamaremos poda a todas las optimizaciones que nos permitan recorrer menos posibles soluciones o evitar recorrer posibles soluciones en su completitud, pero aun así, lograr obtener la respuesta correcta.

Algorithm 2 Backtracking*(const info_t &info, int i, int riesgo, int beneficio, int potencial)*

```
1:  if i ≥ info.cantidad then
2:      if (valido ≠ true) or (riesgo > info.limite) then
3:          beneficio ← 0
4:      end if
5:      mejorBeneficio ← max{beneficio, mejorBeneficio}
6:      return beneficio
7:  end if
8:  if i == 0 then
9:      for j = 0...info.cantidad − 1 do
10:         potencial ← potencial + info.beneficios[i]
11:     end for
12: end if
13: if beneficio + potencial < mejorBeneficio then
14:     return −1
15: end if
16: potencial ← potencial − info.beneficios[i]
17: if riesgo > info.limite then
18:     return −1
19: end if
20: b1 ← Backtracking(info, i + 1, riesgo, beneficio, potencial)
21: if potencial ≠ 0 then
22:     potencial ← potencial − info.beneficios[i + 1]
23: end if
24: b2 ← Backtracking(info, i + 2, riesgo + info.contagios[i], beneficio + info.beneficios[i], potencial)
25: return max{b1, b2}
```

Este algoritmo tiene tres cambios esenciales:

- Primero en la linea (24), vemos que hay un $i + 2$ en comparación al $i + 1$ de la linea (8) del pseudocodigo de fuerza bruta, este pequeño cambio se asegura que no nos fijemos combinaciones donde tengamos dos tiendas consecutivas, reduciendo el espacio de soluciones a las soluciones que son geográficamente validas y solucionando el problema que tiene fuerza bruta.
- Segundo, desde la linea (17) a (19) se verifica que hasta ese punto no se supere el nivel de contagio limite, esto implica que una vez que una combinación de tiendas supera el limite, el algoritmo finalizara e invalidara el camino (devolviendo -1, un valor invalido). Este cambio logra optimizar el algoritmo ya que no perderá tiempo analizando un caso que claramente es innecesario. Esto se reconocerá como la poda por factibilidad.
- El tercer y ultimo cambio lo reconoceremos como poda por optimalidad. Este calcula el posible potencial de un nodo. Cuando vemos la primer tienda, se calcula la suma de beneficios de todas las siguientes tiendas en las lineas (8) a (12), adicionalmente, al final de cada posible caso, guardamos en una variable externa, el mejor beneficio que se encontró (en la linea (5)). De esta forma podemos corroborar que el beneficio actual en conjunto al beneficio potencial supere al mejor beneficio encontrado, si no lo supera, sabemos que no tiene sentidos continuar corriendo esa rama particular. Esta ultima verificación se hace en las lineas (13) a (15). Actualizamos el potencial en la linea siguiente a la verificación, preparándolo para la siguiente tienda. Como ambos llamados recursivos avanzaran a distintas tiendas, entre ambos llamados debemos adaptar el potencial; si este es distinto a cero, sabremos que hay mas tiendas, permitiéndonos acceder a una posición en el arreglo sin superar la posición máxima

La imagen de ejemplo de este algoritmo muestra como el primero de los tres cambios afecta el crecimiento del árbol de posibles soluciones. Eliminando los subárboles inválidos, podemos reducir nuestro espacio de posibles soluciones, particularmente en casos en una avenida larga con muchas tiendas. Estos

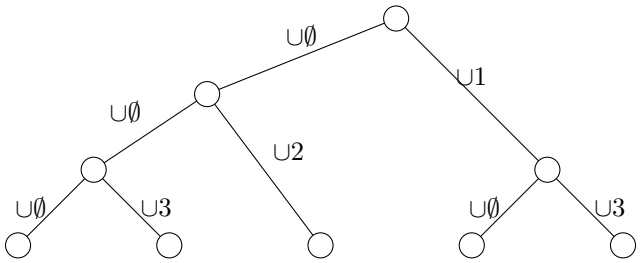

Figura 2: Ejemplo de ejecución del algoritmo de backtracking donde usamos el primer llamado recursivo para ir bajar hacia la izquierda de un nodo y el segundo llamado para bajar hacia la derecha del mismo.

cambios nos garantizan un algoritmo mas rápido ya que recortan los casos innecesarios. Sin embargo, comparte la misma complejidad que fuerza bruta, $O(2^n)$. Esto es por que en el peor caso, ninguna de las podas afecta el recorrido. Seguimos con una complejidad espacial de $O(1)$ ya que aun no utilizamos estructuras adicionales.

2.3. Programación Dinámica:

Programación dinámica divide al problema en subproblemas de tamaños menores, que son mas fáciles de resolver, pero evitando resolver mas de una vez un mismo subproblema en diferentes llamadas recursivas. Para esto mantiene una estructura de datos, donde almacena los resultados que ya han sido calculados hasta el momento, para su posterior reutilización si se repite una llamada a un subproblema. Entonces, cuando se hace una llamada recursiva a un subproblema primero se chequea si este subproblema ya ha sido resuelto. Si lo fue, se utiliza el resultado almacenado. En caso contrario, se realiza la llamada recursiva. Este esquema se llama memoización y es la principal ventaja de estos algoritmos. Esto nos permite ahorrar recursos (principalmente temporales) ya que accederemos a esta estructura con complejidad constante. Para esta implementación optamos por una matriz $\in R^{3*m}$ donde m es el nivel de contagio y dentro de la matriz guardaremos los mejores beneficios para cada nivel de contagio (dependiendo la tienda). Solo observaremos tres tiendas en simultaneo y jamas tendremos que volver a visualizar la información de una tienda anterior, por ende en vez de guardar la información calculada para cada tienda, solo guardaremos la información necesaria de las tres, (por ende R^{3*m}).

Algorithm 3 dinamicPrograming*(const info_t &info)*

1: $memo \leftarrow matriz(nxm)$
2: $dinamicPrograming(info, 0)$
3: **return** $max\_element(memo[(info.cantidad - 1)\,\%3])$

2.3.1. Estructura de memoización:

Como mencionamos anteriormente, la estructura de memoización utilizada para este problema se basa en una matriz $\in R^{3*m}$ y siempre observaremos la i-ésima tienda en conjunto con la $(i-1)$-ésima e $(i-2)$-ésima tienda. Nuestro algoritmo tiene cuatro pasos principales para el uso correcto de la matriz. Asumiendo que estamos en una iteración i-ésima:

- Primero: Limpiamos la j-ésima fila (la fila de la tienda actual). Esta matriz tiene solo tres filas, e iremos rotando la fila actual utilizando un modulo. Pondremos esta fila en cero (como vemos en el algoritmo superior) en la linea (5). Esta fila originalmente tendremos la información de la $(i-3)$-ésima tienda, pero por lo dicho antes esta información ya no sera necesaria, por ende podemos reciclar el espacio para la nueva iteración.
- Segundo: guardamos el beneficio de la tienda. Este es el paso mas simple y consiste en guardar el

Algorithm 4 dinamicProgramingAux*(const info_t &info, int i)*

```
1: if i ≥ info.cantidad then
2:     return
3: end if
4: j ← i %3
5: memo[i] ← {0, ..., 0}
6: memo[j][info.contagios[i]] ← info.beneficios[i]
7: k ← (j + 1) %3
8: for l = 1...info.limite − 1 do
9:     memo[j][l + info.contagios[i]] ← memo[k][l]info.beneficios[i]
10: end for
11: k ← (j + 2) %3
12: for l = 1...info.limite − 1 do
13:     memo[j][l] ← max{memo[j][l], memo[k][l]}
14: end for
15: dinamicProgramingAux(info, i + 1)
```

beneficio en la columna correcta (determinado por el nivel de riesgo del local), en la j-ésima fila. Esto ocurre en la linea (6) de nuestra función auxiliar.

- Tercero: el paso de conexión. Asumiendo que el algoritmo se ejecuto correctamente en la dos filas por arriba de la fila actual (llamaremos k-ésima fila) tenemos los mejores beneficios desde la primer hasta la $(i-2)$-ésima tienda. Este paso se encarga de calcular los posibles beneficios para cada nivel posible de contagio agregando la tienda actual al conjunto de tiendas que generaron los beneficios en la k-ésima fila y guardarlos en la j-ésima fila. Este paso nos permite calcular todos las posibles soluciones para el cual la i-ésima tienda esta contenida.

- Cuarto: Este es el paso mas importante y es el que genera todas las soluciones para la cual la i-ésima tienda no sera incluida en el conjunto de solución. Como el paso anterior combino la tienda actual con la tienda $i-2$ (la ultima tienda con la cual se pudo combinar en el conjunto), lo que hacemos es comparar todos los beneficios que almacenados en la fila de la $(i-1)$-ésima tienda con los de la i-ésima, quedándonos con el beneficio mas grande para cada nivel de contagio. Esto garantiza que la j-ésima fila tendrá todos los mejores beneficios para cada nivel de riesgo, este o no incluida la tienda actual en el conjunto que forma la posible solución.

Siguiendo estos cuatros pasos nos garantiza que progresivamente, tengamos los mejores beneficios para cada nivel de contagio y podemos finalizar nuestro algoritmo buscando el máximo beneficio en la ultima fila operada.

Esta función se llama iterativamente una única vez de cada tienda, y en cada llamado recorremos arreglos de longitud m, con m siendo el limite de riesgo. Por ende tendrá una complejidad temporal de $O(n*m)$, con n siendo la cantidad de tiendas. También, tendremos una complejidad espacial de $O(m)$ por la matriz que tenemos que generar.

3. Experimentación

Para la comparación entre métodos decidimos crear un conjunto de pruebas para correr los distintos algoritmos bajo los mismo parámetros y así poder ver las ventajas y desventajas de cada experimento. Los distintos experimentos son los siguientes:

- Experimento 1: este experimento tendrá todos unos en contagios, y $n+2$ como limite de contagio con n siendo la cantidad de tiendas en la avenida. En este experimento, ninguna combinación de tiendas podrá superar el limite de riesgo, ya que si todas en conjunto fuesen seleccionadas,

sumarían un riesgo de n, sin embargo, esto seria imposible ya que no podemos seleccionar dos tiendas consecutivas. Para los beneficios optamos por valores decrecientes de $n-1$ a 0, generando una fuerte concentración de beneficios al inicio de la avenida tire. Optamos por este arreglo de beneficios ya que permitirá ver bien los beneficios de la poda de optimalidad en backtracking.

- Experimento 2: este segundo experimento es tiene a 5 como limite de riesgo y un vector de 1s en los contagios. Para beneficios, utilizaremos un arreglo de todos ceros. Creemos que de esta forma jamas se podrá utilizar la poda por optimalidad en backtracking ya que el potencial, el beneficio actual y el mejor beneficio encontrado son todos cero, dando la impresión a nuestro algoritmo de que siempre puede haber una mejor combinación.
- Experimento 3: con el tercer experimento queremos ver el efecto de aumentar n, manteniendo un limite de riesgo constante. Optamos por hacer cuatro casos, con el limite teniendo los valores 15, 20, 25 y 30 respectivamente. De esta forma tendremos distintas muestras del mismo experimento.
- Experimento 4: finalmente, nuestro ultimo experimento vera como afecta la variación del limite de contagio, muy similar al experimento 3, tendremos cuatro casos con cuatro n distintos, 15, 20, 25 y 30 nuevamente y el limite incrementando de 2 a 30. Así podremos ver el como esta variable afecta a la programación dinámica.

Notar; los primeros tres experimentos tienen variación de n, este se moverá desde 10 hasta 30 ya que algún valor menor seria muy chico como para notar diferencia alguna y consideramos que 30 seria lo suficientemente grande.

3.1. Análisis:

3.1.1. Fuerza Bruta:

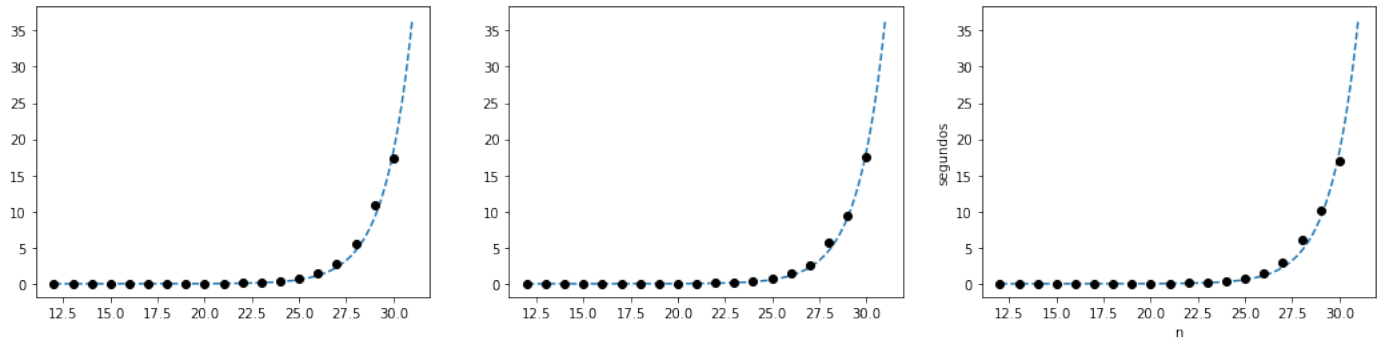

Figura 3: Fuerza bruta sobre el primer, segundo experimento y una muestra al azar.

Viendo los gráficos de todos los experimentos corridos se puede ver que fuerza bruta, al no tener ningún tipo de poda u optimización, su duración incrementa estrictamente pegado a su complejidad temporal de $O(2^n)$. Este incremento parece ser estático entre todos los experimentos. Esto no es ninguna sorpresa, ya que siempre prueba todos las combinaciones que generan en el espacio de soluciones. Dado a que cada tienda puede estar, o no estar en el conjunto, se genera esta complejidad.

Ya que el incremento es tan radical en comparación a los otros métodos (como veremos mas adelante), las lineas de fuerza bruta en los próximos gráficos no serán incluidos por que afectarían las escalas y no se podrían visualizar bien los resultados.

3.1.2. Backtracking:

Para backtracking hay dos experimentos particulares; los primeros dos. El primer experimento genera el peor caso para la poda por factibilidad mientras intenta hacer un buen caso para la poda por optimalidad. Siguiendo los experimentos, vemos que en el primer experimento logra que ningún caso de test pueda alcanzar su limite de riesgo. Recordando la poda por factibilidad, esta poda se encarga de no seguir revisando un camino luego de que haya superado el limite de riesgo. Pero ya que esto jamas ocurre, no debería optimizar el algoritmo en ningún sentido.

El segundo experimento, en cambio, intenta darle la desventaja a la poda por optimalidad que viendo el pseudocodigo en las lineas (13) a (15) podemos ver que esta poda termina una posible solución cuando calcula que el camino actual es peor que el mejor valor obtenido hasta el momento, sin embargo, no termina esa posible combinación si el posible valor es igual. Por esta razón, una avenida con todos beneficios nulos es la mejor forma de evitar optimizaciones de la poda. Al no tener beneficios, el mejor beneficio siempre va a ser cero y el posible valor de cada combinación también lo sera, por ende, se anula el efecto de la poda.

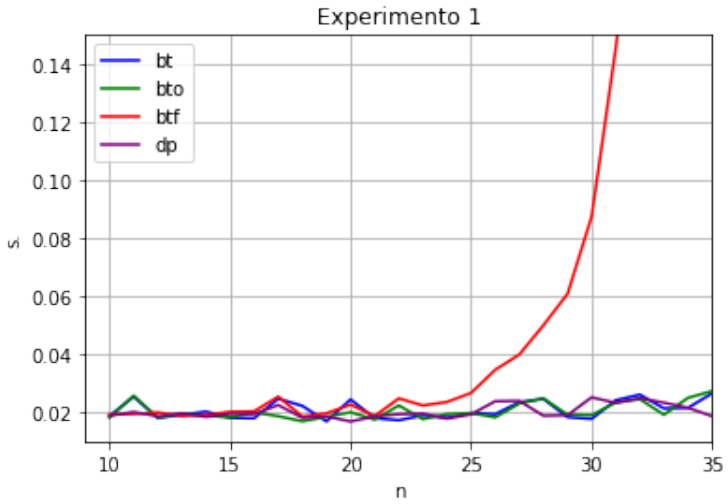

Figura 4: Resultados del primer experimento donde se mide cuantos segundos tarda en correr los algoritmos con una cantidad variada de tiendas.

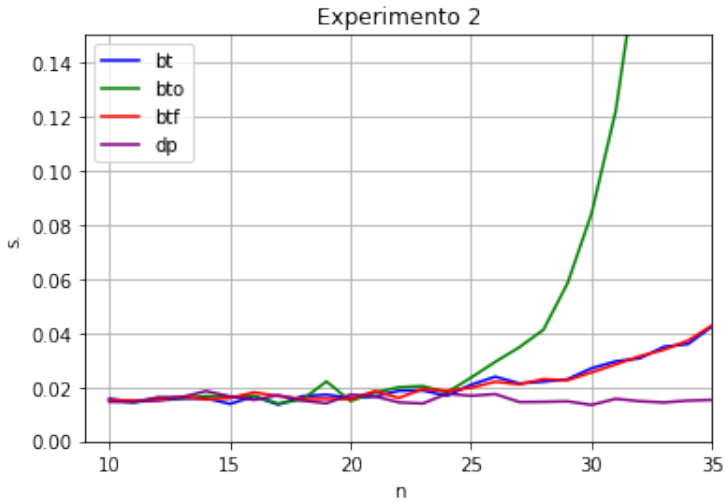

Figura 5: Resultados del segundo experimento donde se mide cuantos segundos tarda en correr los algoritmos con una cantidad variada de tiendas.

Como se ve en los graficos anteriores, en el primer experimento, la poda de factibilidad (**btf**) claramente no optimiza al algoritmo, pues como se planteo antes, es un caso diseñado para que la poda jamas

se aplique, sin embargo, la poda de optimalidad (**bto**) logra negar (en el rango visto) todo crecimiento notorio del método.

El experimento 2 muestra, claramente, lo contrario. Acá se ve la importancia de ambas podas ya que una única poda no logra optimizar todos los casos. La poda por factibilidad evita que el algoritmo no vea casos que sabemos que no pueden ser solución, mientras el de optimalidad evita que busquemos casos que sabemos que no tienen el potencial para ser la mejor solución.

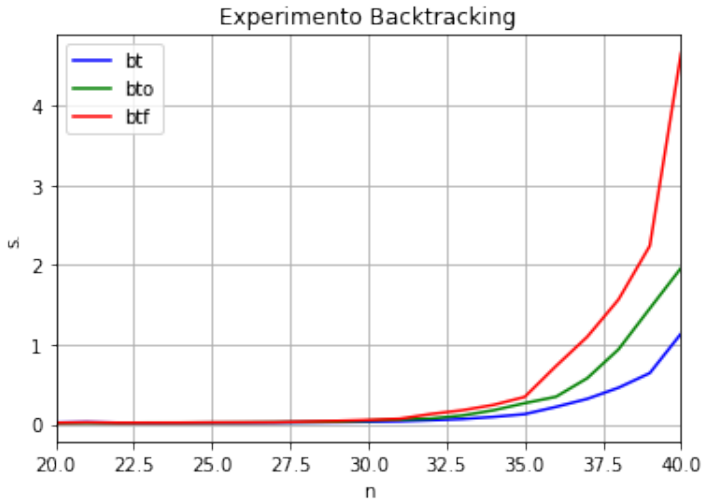

Figura 6: Resultados de un experimento donde se muestran las distintas podas con distintos efectos.

Ahora que vemos un experimento mas generico, podemos ver que ambas podas optimizan de diferente manera sin embargo el algoritmo general de backtracking (**bt**) que contiene ambas podas, es muchísimo mas eficiente que ambas podas por separado. Notemos igual que sin ninguna poda, este algoritmo es idéntico a fuerza bruta, por lo que si se conoce alguna forma de podar el espacio de soluciones, este siempre sera mejor que fuerza bruta estándar.

3.1.3. Programación Dinámica:

Para el caso de programación dinámica se decidio analizar de que manera se ve afectado su tiempo de ejecución en función de n y el limite de riesgo para los cuales fueron testeados, por el experimento 3 y experimento 4 respectivamente, donde una de las variables se mantenía constante mientras la otra variaba.

Se utilizo una escala mucho mas grande ya que el escalado era mucho menos prominente que en backtracking o fuerza bruta. Veamos el gráfico del experimento 3 donde puede observarse como el incremento de locales afecta la cantidad de segundos que tarda este algoritmo. Estos experimentos solo fueron corridos con programación dinámica ya que la complejidad de los otros algoritmos no lo permitían.

Volviendo a ver el gráfico del experimento 1 y del experimento 2 se puede ver como en el peor caso el algoritmo de backtracking llegan a $0{,}04$ segundos al rededor de $n = 28$, mientras que en gráfico del experimento 3 se observa que programación dinámica llega $0{,}04$ segundos luego de $n = 1,000$. La razón de esto es clara una vez que se conocen las complejidades de cada algoritmo.

Luego, el gráfico del experimento 3 analiza el crecimiento dado que el limite de riesgo aumenta. Nuevamente es lineal dado que la cantidad de locales no aumenta y aunque aumente, es muy gradual y el valor de limite debe alcanzar números extremadamente altos para superar al algoritmo de backtracking con una cantidad baja de tiendas.

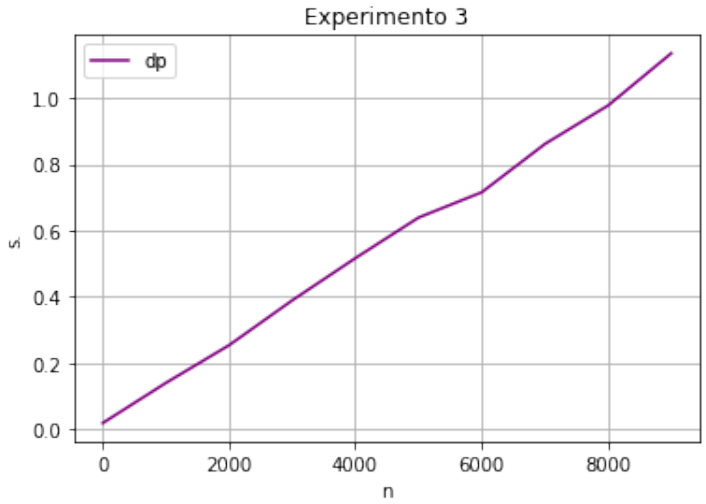

Figura 7: Resultados del tercer experimento donde se mide cuantos segundos tarda en correr el algoritmo variando la cantidad de locales.

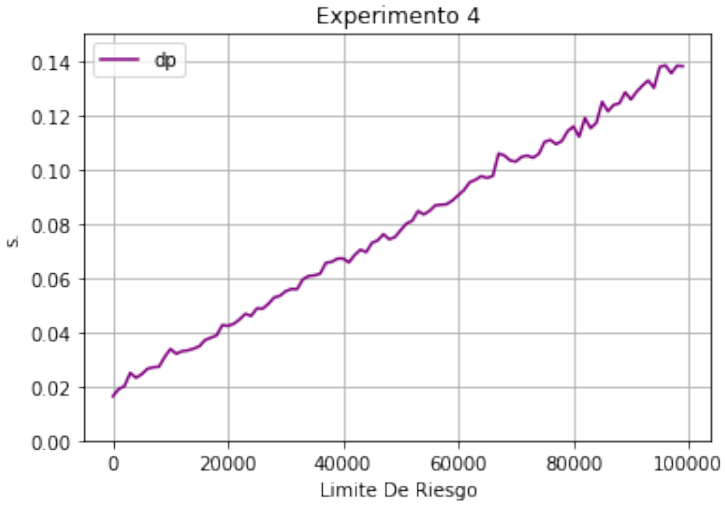

Figura 8: Resultados del cuarto experimento donde se mide cuantos segundos tarda en correr el algoritmo con limite de riesgo variado.

4. Conclusión

Como vimos en la sección anterior, para este problema, el mejor algoritmo era sin duda el implementado utilizando programación dinámica, sin embargo, esto no siempre es así. Aunque tuvo el crecimiento mas leve, fue el único en tener complejidad espacial. Adicionalmente, en este problema pudimos mantener esta complejidad espacial de manera lineal. Hay casos donde esto no es posible y podemos tener complejidad espacial y temporal cuadrática, logrando que programación dinámica pueda llegar a ser mas costosa que backtracking en todo sentido. Ambos métodos son muy útiles y muy eficientes pero solo cuando se implementan correctamente, eligiendo podas correctas o la estructura correcta para memoizacion para backtracking y programación dinámica respectivamente. En caso de que esto no sea posible, siempre se puede recurrir a fuerza bruta, que aunque no pueda aportar nada mejor que estos dos métodos, es el mas fácil y rápido de implementar.