TP1 TD5

Tomás Gallo, Guadalupe Molina, Jazmín Sneider

Abril 2024

1 Introducción

En este informe se detalla la implementación PWL, una forma de aproximar una serie de puntos mediante el uso de funciones continuas a trozos, específicamente, funciones continuas piecewise lineal (PWL). El propósito principal radica en identificar aquellos puntos (breakpoints) que, al unirse a través de funciones lineales, minimicen el error total desde el primer punto hasta el último ingresado por el usuario. Para lograr este objetivo, se emplean tres enfoques distintos:
El primer enfoque se basa en una estrategia de **fuerza bruta**, donde se intenta encontrar la mejor solución explorando todas las posibles combinaciones de puntos. El segundo enfoque, basado en **Backtracking**, busca mejorar el algoritmo de fuerza bruta al implementar técnicas de poda que permiten descartar varias opciones, reduciendo así la carga computacional. Estos dos primeros son recursivos. Por último, se recurre a la **programación dinámica**, aprovechando el uso eficiente de la memoria y la repetición de casos, lo que posibilita obtener la solución de manera considerablemente menos compleja desde el punto de vista computacional en comparación con los otros dos enfoques. Este difiere de los anteriores por ser iterativo.
A través de este informe, se explorará la implementación de cada uno de estos enfoques, así como su rendimiento y eficacia en la aproximación de la serie de puntos dada. También compararemos los lenguajes de programación que fueron usados para implementar las funciones, *python* y *c++*.

2 Implementación:

Optamos por iniciar el desarrollo de las funciones en Python, dado que fue el último lenguaje que utilizamos en la materia anterior y, por lo tanto, teníamos su sintaxis más fresca en nuestra memoria. Además, Python ofrece varias ventajas en comparación con otros lenguajes, especialmente en lo que respecta a la sintaxis de listas y la flexibilidad en la cantidad de parámetros que se pueden devolver por función. Esta característica resultó especialmente útil para abordar la primera parte del proyecto de manera más eficiente y rápida. Definimos que todas las funciones tomarían por parámetro los siguientes elementos:

- *string* datos: el nombre del archivo json que contiene los puntos a analizar

- *int* breakpoints: representa la cantidad de puntos que contendrá la solución.
- *int* m1 y *int* m2: representan las dimensiones de la grilla en x y en y respectivamente.

Decidimos también que cada función principal sólo se encargaría de procesar los datos que pasó el usuario (leer el archivo y definir las grillas) para luego llamar a una función auxiliar con parámetros extra que la ayudarían a encontrar la solución óptima con los datos ya procesados y variables para ir rellenando a lo largo de su trabajo. En los primeros dos algoritmos la función llama a la auxiliar considerando como si hubiera un nodo inicial (fuera del plano marcado por las grillas) desde donde se inicia el proceso. En programación dinámica esto cambia pero se explicará más abajo. Establecimos el formato de la solución como una lista de tuplas, donde cada tupla representa las coordenadas de un breakpoint en ambos ejes, x e y. Además, las funciones proporcionanan el error junto con la solución para que el usuario pueda visualizarlo fácilmente.

2.1 Fuerza Bruta

Python: En nuestro enfoque de fuerza bruta, la función principal le pasa a la auxiliar una solución inicial vacía y un error muy grande, que luego se compara con los errores de las soluciones que se van generando. En el código, para cada posición en el eje x de la grilla definida previamente, consideramos todas las opciones posibles. Este proceso implica evaluar dos escenarios principales:

1. **Seleccionar un punto en el eje *y*:** Para cada posición x en la grilla, exploramos todas las posibles coordenadas en el eje y. Esto significa que para cada x, evaluamos todos los puntos disponibles en el eje y como posibles breakpoints.
2. **No seleccionar ningún punto (opción adicional):** Además de considerar puntos específicos en el eje y, también tenemos en cuenta la posibilidad de no seleccionar ningún punto en absoluto. Esta opción adicional permite que la cantidad de breakpoints no sea necesariamente igual al tamaño de la grilla x, siempre y cuando se cumpla la restricción de no exceder la cantidad de posiciones disponibles.

De esta manera, evaluamos todas las combinaciones posibles de tamaño **k** (donde k es la cantidad de breakpoints proporcionados como parámetro). Durante este proceso, implementamos casos base para detener la recursión y descartar soluciones incorrectas, como aquellas cuyo tamaño no coincide con la cantidad de breakpoints solicitados, o que no comienzan y terminan en las posiciones adecuadas en el eje x, o que presentan repeticiones de coordenadas x.

Cada vez que una combinación de breakpoints cumple con los criterios establecidos y termina en la posición correcta, la función ErrorSolucion calcula su error y determina si esa solución tiene un error menor que la óptima hasta el momento. Si ocurre esto, la solución recién encontrada se guarda como la óptima y su error se convierte en el error mínimo hasta el momento. Al final

de todas las recursiones, la solución que acumuló el menor error es la que se conserva.

Es importante destacar que conforme aumenta el tamaño de la entrada, el tiempo de ejecución de este enfoque también aumenta, ya que se deben explorar todas las combinaciones válidas y calcular el error de cada una de ellas.

2.2 Back Tracking

Python: La implementación del algoritmo Backtracking se basa en el mismo enfoque que busca todas las posibles soluciones a un problema. Sin embargo, para optimizar este proceso y evitar el desperdicio de tiempo en soluciones que no conducen a resultados válidos, se aplican diversas técnicas de poda.

1. La primera de estas técnicas es la poda por factibilidad. Después de la primera iteración, si se identifica que la lista solución ya no contiene al primer elemento de la grilla x, se descarta la continuación de la exploración de esa solución, ya que se determina que no puede conducir a una solución válida.

2. Otra poda es la que tiene en cuenta la acumulación de error en la solución parcial hasta el momento. Si esta solución parcial ya acumula más error que la solución óptima guardada, se detiene la exploración de esa opción, ya que se concluye que no puede conducir a una solución más óptima. Esta poda es por optimalidad

3. Finalmente, se incorpora una poda por factibilidad, que se aplica cuando se identifica que el número de breakpoints excede el espacio disponible en la grilla x. Dado que no se puede colocar más de un breakpoint por punto en x, la exploración de esa solución se detiene.

Estas técnicas de poda permiten reducir, en la mayoria de casos, significativamente el tiempo de ejecución del algoritmo, lo que se refleja en un rendimiento bastante mejorado conforme aumenta el valor de los parámetros. De igual manera, sospechamos hay podas mucho más eficientes que otras, así que a continuación experimentaremos para ver cómo cada poda mejora el rendimiento de backtracking.

1. Para la primera y segunda poda tomaremos la instancia $titanium.json$ y los parametros breakpoints=4, m1=6 y m2=6. Las vamos a sacar, de a una y dejando las otras dos, para ver cómo cambia la ejecución.

2. Para la tercera poda tomaremos la instancia $titanium.json$ y los parametros breakpoints=6, m1=6 y m2=6 (cuando breakpoints=m1 es donde mejor funciona la poda). La vamos a sacar para ver cómo cambia la ejecución.

Fuerza Bruta demora con los mismos parámetros 0.1469 segundos

	Backtracking s/la poda	Backtracking c/la poda
Poda 1:	0.02177 segundos	0.02153 segundos
Poda 2:	0.02156 segundos	0.0067 segundos

Figure 1: La primera no influye tanto, la segunda es muy efectiva

Fuerza Bruta demora con los mismos parámetros 1.086 segundos

	Backtracking s/la poda	Backtracking c/la poda
Poda 3:	0.0256 segundos	0.006 segundos

Figure 2: Así comprobamos que las mejores podas son la segunda y la tercera

2.3 Programación Dinámica

Python: Para abordar este problema, optamos por utilizar un enfoque Bottom-Up en el desarrollo de nuestro algoritmo de programación dinámica. Para evitar la repetición de casos, implementamos un superdiccionario de memorización que registra los resultados de los problemas más pequeños y los utiliza para resolver los más grandes.

Nuestro enfoque se basa en la construcción de errores a partir del error anterior (que tiene un error acumulado) más el error al unir el punto donde se encuentra el último error para llegar al punto actual. Es decir, para buscar un error mínimo, hay que buscar el mínimo anterior y sumarle lo que nos cueste llegar al punto nuevo . Entre estas sumas del error anterior más el actual, eventualmente se buscará el mínimo para así ir generando tanto el camino como el resultado óptimo.

En la construcción de la solución, cada nuevo punto en el diccionario no solo tiene asociado su error, sino también los pares (x, y) que indican las posiciones de la grilla donde se encuentran los puntos anteriores del camino mínimo. Esto nos permite conservar los puntos óptimos a medida que generamos el error mínimo.

Finalmente, utilizamos estos pares (x, y) guardados para asociarlos a un valor específico dentro de la grilla_x.

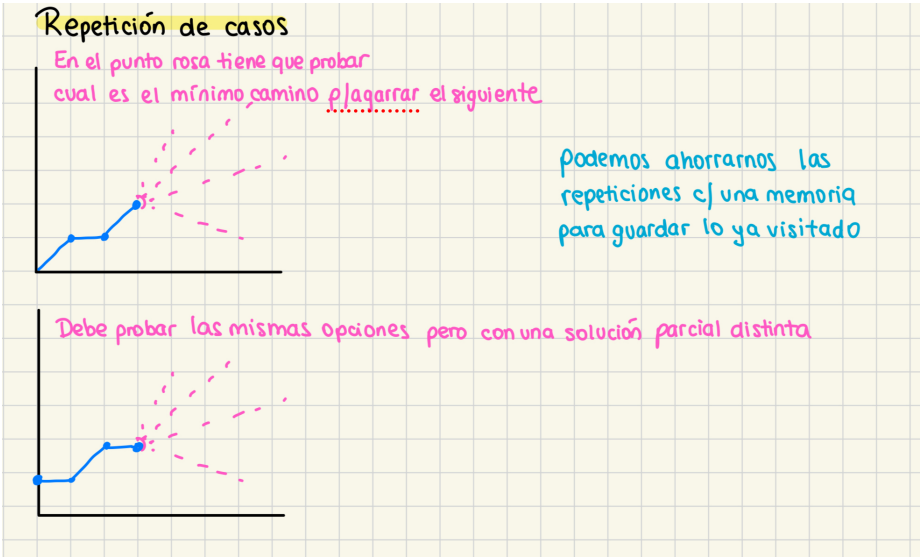

Figure 3: ¡Repetición de Casos!

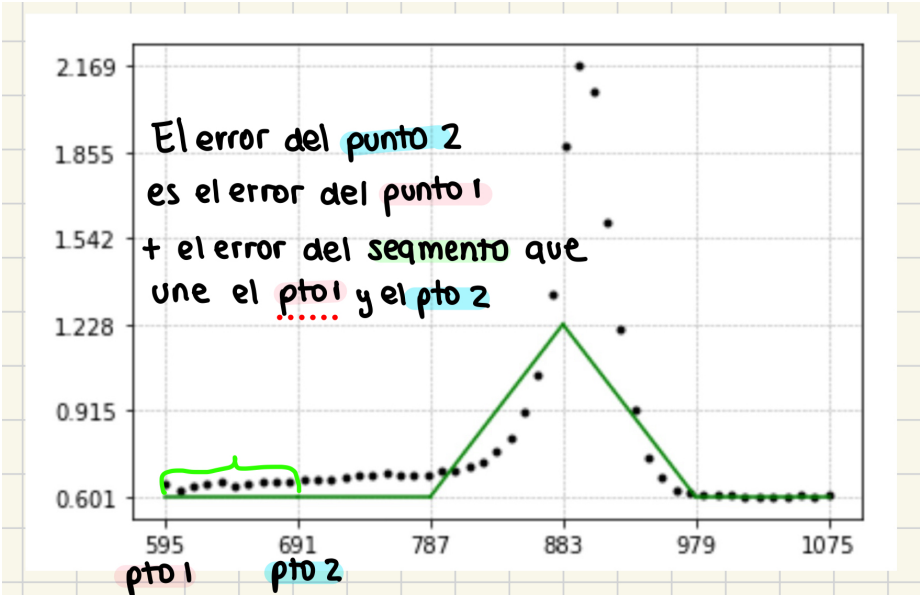

Figure 4: ¿Cómo pensamos PD?

La función principal procesa los datos como de costumbre, pero ahora también inicializa el superdiccionario. Este superdiccionario tiene como claves una tripla de enteros, representando la posición x, la posición y y la cantidad de breakpoints realizados hasta el momento. El valor asociado a cada clave es una lista que contiene el error y una lista de tuplas que devuelven ese error.

En la función principal breakpointsPD, la función inicializar se encarga de llenar todos los lugares (breakpoints, m1, m2) con error infinito y listas vacías. Luego, breakpointsPD llama a una función auxiliar que realiza todo el trabajo de buscar la solución óptima como explicamos anteriormente.

3 Experimentacion:

(a) **Calidad:** La calidad de las aproximaciones está fuertemente influenciada por los parámetros que el usuario proporciona en los breakpoints, m1 y m2. Estos parámetros juegan un papel crucial en la calidad de la aproximación resultante. En cuanto al tamaño de la grilla (m1*m2), su elección es importante si la intención es disminuir el error. Optar por una grilla con un rango mayor, aunque aumente el tiempo de ejecución debido a la mayor cantidad de posibilidades, permite disponer de más puntos para seleccionar como breakpoints. Esto facilita capturar una mayor cantidad de puntos cercanos a los puntos originales, lo que resulta en una mejora en la precisión de la aproximación. Por el contrario, si se eligen grillas con pocos puntos, existe el riesgo de perder detalles importantes en los datos. Por tanto, es conveniente elegir puntos razonables para ambos valores m1 y m2, teniendo en cuenta la cantidad de puntos que ingresan en el archivo. Para lograr error 0, la grilla x debería tener igual cantidad de posibilidades que puntos ingresados, y la grilla y debería contener todas las alturas de los puntos. Sin embargo, es importante considerar que a medida que aumenta la cantidad de puntos, se vuelve computacionalmente inviable esperar que se prueben todas las posibilidades, especialmente cuando hay un gran número de puntos. Por ejemplo, en el caso de la instancia "optimistic_instance", donde hay muchos puntos, el error tiende a ser grande en los casos donde podemos probar. Sin embargo, en otros casos, como en "titanium" o "aspen_simulation", al elegir puntos adecuados, podemos llegar a errores más pequeños y que corren en un tiempo razonable de esperar. Es importante destacar que m1 también limita el máximo de breakpoints que podemos añadir. Por lo tanto, a menor tamaño de grilla x, menos segmentos podremos añadir como máximo para aproximar la función. Esto significa que la elección de m1 también debe ser cuidadosa y considerada en función de la precisión deseada en la aproximación.

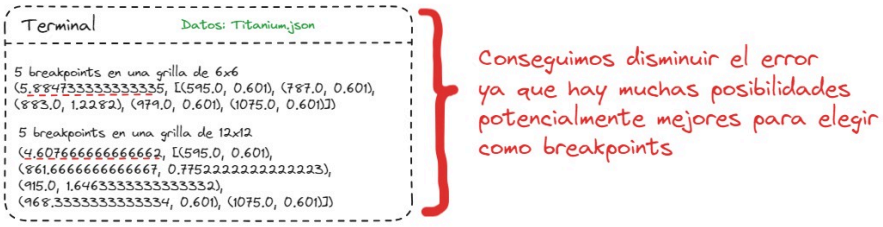

Figure 5: Aumentar la grilla reduce significativamente el error

La cantidad de segmentos también debe ser elegida inteligentemente ya que con más segmentos, se puede reducir el error de aproximación al adaptar mejor la función aproximada a los puntos dados. Cada segmento puede ajustarse individualmente para minimizar el error en su región específica, lo que puede conducir a una aproximación más precisa en general.

Por último, con respecto a la calidad también nos parece importante aclarar que el error también depende de la distribución que tiene cada conjunto de puntos, ya que depende de la posición en el eje "y". Por esto es que algunos datos, como ethanol_water_vle.json, teniendo más puntos que aspen_simulation.json permite obtener un error mucho menor con la misma cantidad de breakpoints, m1 y m2. La forma de los datos ethanol es más aproximable por el algoritmo gracias a su distribución similar a una lineal.

(b) **Pasaje a C++:** La consigna pedía resolver el problema en dos lenguajes diferentes, Python y C++. Al iniciar, decidimos abordar la resolución con Python, principalmente por la frescura de nuestra memoria con la sintaxis y una serie de ventajas, especialmente en cuanto al manejo de variables, que proporciona este lenguaje. Por ejemplo, en Python, las estructuras de datos como listas (*list*) y tuplas (*tuple*) nos permitían acceder fácilmente a sus elementos utilizando la sintaxis de corchetes [*posición*]. Además, Python nos brindó una herramienta extremadamente útil para los llamados recursivos: la capacidad de pasar una copia modificada de la lista *solucion* a una función recursiva. El siguiente comando nos permitió mantener varias copias de la solución en diferentes etapas de la recursión.

```
solución+[(grilla_x[x],grilla_y[j])]
```

Python también nos dió acceso a una serie de funciones integradas extremadamente útiles, como **max**, **min**, **len**, **in** y **np.linspace** que facilitaron enormemente nuestra tarea y nos permitieron avanzar en la programación sin tener que preocuparnos por implementar estas

funciones por separado. Finalmente, otra gran ventaja de Python fue su facilidad para imprimir resultados. La función print nos permitió mostrar listas, tuplas y otras variables de manera directa y sencilla, sin necesidad de utilizar ciclos o comandos adicionales, como sí tuvimos que hacer en C++. Nos permitió probar nuestro código de manera instantánea para encontrar errores con mayor velocidad. Cuando tuvimos que traducir el código a C++, inicialmente pensamos que el proceso sería bastante rápido, ya que solo teníamos que ajustarnos a la sintaxis, mientras que la lógica de programación seguía siendo la misma. Sin embargo, nos encontramos con varios desafíos que complicaron nuestra tarea:

- Al analizar cómo pasar la función $armar_grilla$, nos dimos cuenta de que teníamos que construirla completamente desde cero, ya que no teníamos acceso a herramientas como np.linspace. Aunque no resultó ser una tarea extremadamente complicada, requirió algunos cálculos extra y líneas de código adicionales.
- Utilizamos como equivalente al tipo *List* en python, el tipo *Vector* en c++, que afortunadamente se manipulan de manera similar con leves modificaciones en la sintaxis.
- Acceder a los elementos de las tuplas no era tan sencillo como en Python. Tuvimos que recordar cómo usar iteradores y la función *get* para poder manipular los elementos correctamente.
- La función *in* para determinar la presencia de un elemento en una lista, no estaba definida en c++ por lo que también tuvimos que implementar esta funcionalidad por separado.
- Uno de los desafíos más significativos fue resolver la ausencia del comando que mencionamos anteriormente para pasarle a la función recursiva una copia de la lista *solucion* con una nueva tupla, como lo habíamos hecho en Python. Tras un tiempo de reflexión, encontramos que podíamos lograr un comportamiento similar utilizando $solución.push\_back(latuplanecesaria)$ antes de la recursión y pasarle *solución* modificada. Como no se estaba pasando una copia y estaba trabajando con la misma variable, después de la recursión había que deshacer la acción con $solución.pop\_back()$, lo que nos permitió manejar todos los casos correctamente.
- La transición a C++ desde programación dinámica resultó ser la menos complicada, especialmente porque anteriormente la mayor dificultad residía en asegurarnos de actualizar correctamente las variables en los casos recursivos. Al abordar el problema con un enfoque iterativo, no tuvimos que enfrentar este desafío. Además, en este punto ya estábamos familiarizados con el manejo de tuplas y diccionarios de una manera diferente, lo cual facilitó aún más la transición.

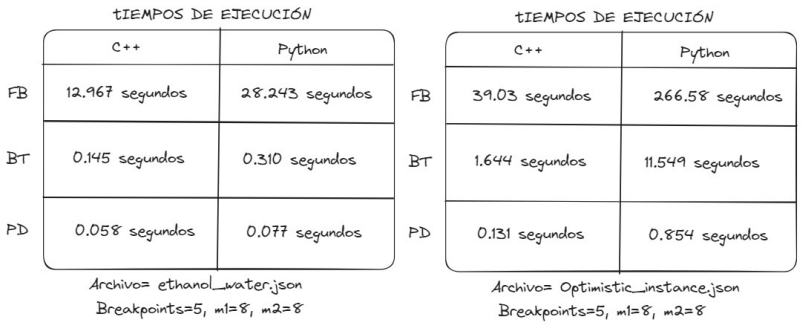

Figure 6: Notable diferencia entre ambos lenguajes

- Por último, una dificultad que ralentizó significativamente nuestras pruebas de casos de test fue la falta de una forma rápida de imprimir vectores en C++. A diferencia de Python, donde esto era instantáneo y nos permitía verificar rápidamente si nuestras funciones estaban generando o modificando las listas correctamente, en C++ tuvimos que recurrir a la creación de variables de tipo vector y utilizar bucles "for" en el "main" para ver cada uno de sus elementos. Aunque no fue un proceso extremadamente complicado, sí resultó considerablemente más lento que con Python.

Después de experimentar con nuestras funciones en ambos lenguajes, nos dimos cuenta de algo interesante: **C++** ofrece un tiempo de ejecución mucho más rápido que **Python**. La diferencia radica en cómo manejan el código: **Python** es un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea en tiempo de ejecución. Por otro lado, **C++** es un lenguaje compilado, lo que implica que el código se traduce completamente a lenguaje de máquina antes de la ejecución, lo que puede resultar en tiempos de ejecución considerablemente más rápidos. Esta diferencia en la forma en que se procesa el código contribuye a que **Python** sea generalmente más lento que **C++** en términos de tiempo de ejecución, especialmente en programas que requieren un alto rendimiento computacional.

Al final del día, se reduce a lo que cada uno valora más: ¿la velocidad y eficiencia de C++, o la comodidad y flexibilidad de Python? Cada programador debe elegir si prefiere la comodidad y flexibilidad que ofrece Python, a pesar de un tiempo de ejecución más largo, o la rapidez de ejecución garantizada al optar por C++ (obviamente también dependiendo de sus habilidades con la sintaxis de cada lenguaje).

(c) **Performance y Propuesta:** Para poder elegir cuál es el algoritmo que propondríamos como mejor y en qué lenguaje nos parece mejor resolver el problema, decidimos realizar una serie de experimentos para sumar a las conclusiones que ya teníamos previamente. Este experimento lo realizamos en dos computadoras distintas: una Macbook Air 2022 y una Dell Inspiron 3581 por si las dudas llegábamos a conclusiones distintas dependiendo de la compu en lo que lo ejecutábamos. Sin embargo, aunque los tiempos de ejecución fueron distintos en ambas compus, la magnitud en la diferencia que se generaba al cambiar ciertos parámetros fue la misma. De igual manera, en los gráficos a continuación, mostraremos la performance en ambas compus.

   i. **Variación de Breakpoints:** Decidimos dejar, en esta parte, x e y fijos y sólo aumentar los breakpoints para ver cómo aumentaba el tiempo de ejecución en los 3 algoritmos. Como es de esperarse, añadir más breakpoints aumenta el tiempo de ejecución en los 3 casos, pero en magnitudes muy distintas entre sí. Llegamos a las siguientes conclusiones:

   - **Con muy pocos breakpoints, fuerza bruta es la ganadora**: Su ventaja radica en que las combinaciones a probar no son demasiadas, lo que implica un menor gasto de tiempo en su exploración exhaustiva. Por otro lado, Programación Dinámica se ve desfavorecida en este contexto debido al tiempo empleado en la construcción y actualización del diccionario, así como en la reconstrucción de soluciones a partir de él. Por su parte, Backtracking también pierde eficiencia al dedicar más tiempo a verificar las condiciones de las podas en lugar de generar directamente las soluciones.
   - **A más breakpoints, gana programación dinámica:** Su eficacia se debe a que evita la repetición de casos al calcular y almacenar los resultados una sola vez en el superdiccionario, ahorrando tiempo. Por otro lado, tanto el enfoque de fuerza bruta como el de backtracking, a pesar de posibles podas en este último, terminan repitiendo cálculos.
   - **El punto anterior aplica siempre que la cantidad de breakpoints sea menor al tamaño de la grilla_x:** Cuando esta condición no se cumple, backtracking se convierte en ganador gracias a una de sus podas. Esta poda evita seguir completando la solución si la cantidad de breakpoints restantes

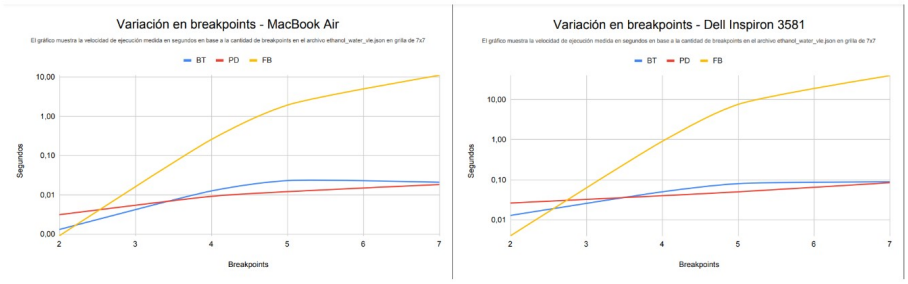

Figure 7: Crecimiento del tiempo de ejecución al aumentar breakpoints

es menor que los lugares disponibles en la grilla. Esto permite que backtracking corte rápidamente las posibilidades y supere tanto a fuerza bruta como a programación dinámica en eficiencia.
CASO ESPECIAL: Hay casos excepcionales como pasa con el archivo "*optimistic_instance.json*", donde backtracking, a pesar de aplicar la poda mencionada, pierde por casi el doble contra programación dinámica. Esto se debe a que, debido al gran número de puntos en el archivo, backtracking (por como está programado) también se encarga de calcular el error total de todos los segmentos para todos los puntos varias veces (a medida que va encontrando soluciones factibles). Este trabajo adicional termina por hacer que backtracking sea menos eficiente que la programación dinámica en este caso particular.

ii. **Variación en dimensión de grilla X:** Manteniendo fijos los breakpoints en un valor pequeño y el tamaño de la grilla Y constante, observamos lo siguiente:
   - Inicialmente, la estrategia de fuerza bruta gana cuando la dimensión de la grilla X es pequeña, dado que, como mencionamos previamente, funciona mejor con un bajo número de breakpoints. Sin embargo, el tamaño de la grilla X tiene un impacto significativo en el tiempo de ejecución tanto de fuerza bruta como de backtracking, ya que sus recursiones están directamente influenciadas por el incremento de su tamaño. Por lo tanto, a medida que aumentamos el tamaño de la grilla X, programación dinámica termina siendo la ganadora en todos los casos. Esto se debe a su eficacia para evitar la repetición de casos y su habilidad para optimizar el proceso al guardar y reutilizar resultados previamente calculados.

iii. **Variación en dimensión de grilla Y:** La variación en la dimensión de la grilla Y, manteniendo fijos los breakpoints y el

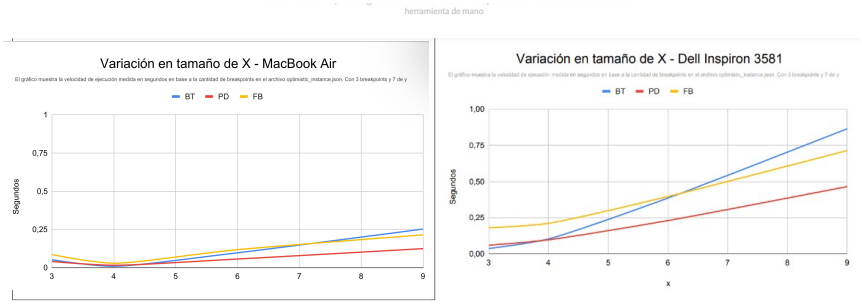

Figure 8: Crecimiento del tiempo de ejecución al aumentar X

tamaño de la grilla X, revela que este parámetro es el que tiene el menor impacto en el tiempo de ejecución en comparación con los otros tres parámetros de entrada de la función (datos, breakpoints, tamaño de la grilla X). En el caso de la estrategia de fuerza bruta, la variación en la dimensión de la grilla Y sí genera un cambio significativo, ya que implica un aumento en el número de iteraciones en el bucle que recorre la grilla X. Esto conlleva a un incremento en el tiempo de ejecución, especialmente cuando la grilla Y es grande, ya que fuerza bruta debe probar todas las combinaciones posibles sin aplicar podas. Sin embargo, para backtracking y programación dinámica, la variación en la dimensión de la grilla Y no tiene un impacto tan significativo en el tiempo de ejecución debido a la capacidad de backtracking para aplicar una poda efectiva mientras que la programación dinámica mantiene su eficacia independientemente de la dimensión de la grilla Y. En general, backtracking salió ganador en nuestros casos de prueba ya que elegimos breakpoints y tamaño de grilla del mismo tamaño y, como explicamos antes, esto le daba una ventaja ante programación dinámica. Sin embargo perdió contra programación dinámica cuando aumentamos mucho la Y, y ademas usamos optimistic_intance.json como datos. Esto ocurre otra vez, porque debe calcular error para la enorme cantidad de puntos varias veces.

4 Recomendación personal:

Después de completar la experimentación y reflexionar sobre nuestro trabajo práctico, hemos llegado a la conclusión de que el mejor algoritmo para resolver este problema en general es la programación dinámica. Observamos que devuelve la misma solución que el backtracking y la fuerza bruta, pero en un tiempo de ejecución casi siem-

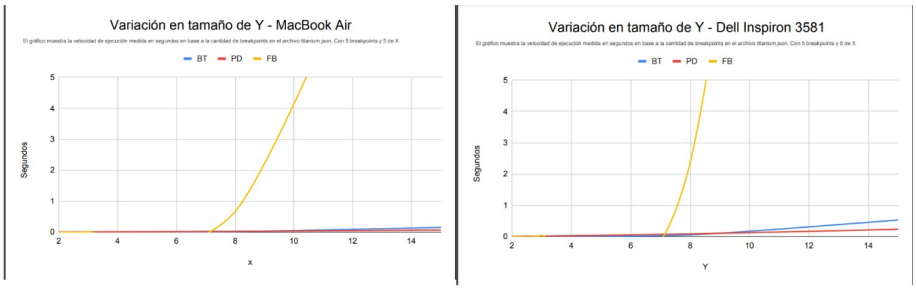

Figure 9: Crecimiento del tiempo de ejecución al aumentar Y

pre menor, y en muchos casos, con una diferencia considerable.

En cuanto a la implementación, aunque la programación dinámica no sigue un enfoque recursivo y requirió que la abordáramos desde cero, sentimos que tuvimos un mayor control sobre el flujo del programa. En comparación con la recursión, donde debemos tener cuidado constante con la actualización de variables, entre otras consideraciones, la programación dinámica nos permitió una estructura más clara y controlada. Por lo tanto, este sería el algoritmo que recomendaríamos para abordar este problema.

En cuanto al lenguaje de programación, al respetar nuestra decisión de recomendar la programación dinámica, encontramos que no hay una diferencia significativa en el tiempo de ejecución entre Python y C++. Sin embargo, observamos que en Python se ahorra más tiempo debido a la flexibilidad en el manejo de tipos de datos y a la facilidad de depuración mediante la inserción de impresiones (especialmente dentro de tuplas y listas). Esto facilita el proceso de depuración en caso de errores. Por lo tanto, recomendaríamos abordar la programación dinámica en Python.

Por otro lado, si se desea utilizar alguno de los otros dos algoritmos, a pesar de las dificultades en la prueba y las limitaciones en los tipos de datos, recomendamos encarecidamente el uso de C++. La diferencia en los tiempos de ejecución entre los lenguajes es significativa. Sin embargo, la elección final depende de las preferencias y habilidades individuales de cada persona.