UNIVERSIDAD TORCUATO DI TELLA
TD V: Diseño de Algoritmos

TP 1: Aproximación de datos vía funciones lineales continuas a trozos

Integrantes:
Lara Barijhoff
Camilo Suárez
Caterina Villegas
Valentina Vitetta

22 de Abril del 2024

Contents

1 Introducción

En este Trabajo Práctico desarrollaremos cuatro algoritmos para afrontar el problema de la aproximación de datos mediante el uso de funciones continuas PWL. El uso de funciones para aproximar datos es esencial para comprender y modelar fenómenos complejos. Sin embargo, varias veces sucede que los datos vienen de funciones desconocidas con comportamientos erráticos e impredecibles. Por esta razón, utilizaremos funciones PWL, las cuales dividen el dominio en segmentos y aproximan los datos con una función lineal en cada uno de los mismos. Nuestro objetivo es desarrollar algoritmos que, dado un conjunto de puntos (x_i, y_i), con i = 1, ..., n, provenientes de una función g(t) desconocida, una discretización (t_i, z_j), con i = 1, ..., m1 y j = 1, ..., m2, y un valor K para la cantidad de breakpoints, encuentren la mejor función PWL que minimize el error total de la aproximación.

Usaremos cuatro enfoques distintos para abordar el problema planteado: fuerza bruta, backtracking, programación dinámica top-down y programación dinámica bottom-up. También propusimos un refinamiento del algoritmo de programación dinámica utilizando teoría de grafos para ahorrarnos aún más cálculos innecesarios. La elección del mejor método a usar puede ser compleja y no siempre evidente ya que puede depender de las características específicas del problema, como los valores de los parámetros elegidos y el tamaño de las instancias. Por esta razón, analizaremos los algoritmos implementados con el fin de examinar y comparar sus fortalezas y debilidades mientras se resuelve el problema planteado.

2 Implementación

2.1 Fuerza bruta

Para el algoritmo basado en Fuerza Bruta, decidimos explorar todas las combinaciones posibles de K breakpoints dentro de la discretización definida por m1 y m2 y así encontrar la mejor aproximación que minimice el error total.

Para ello, realizamos llamados recursivos hasta llegar a la cantidad de breakpoints deseada. Allí, primero evaluamos si la función es factible, es decir, si el primer breakpoint se encuentra en la primera columna de la grilla de la discretización y si el último breakpoint está en la última columna. Al mismo tiempo, utilizando una función auxiliar, verificamos si el error total de la aproximación actual es menor que el mínimo error encontrado hasta el momento. De cumplir ambas condiciones, reemplazamos la mejor solución por la actual.

A fin de generar combinaciones que realmente sean funciones, es decir, que dos breakpoints no

se encuentren en la misma columna, consideramos la abscisa del último breakpoint para comenzar a partir de la siguiente columna. En caso de no haber generado ningún breakpoint, comenzamos a explorar las combinaciones a partir de la primera columna. Por otro lado, debido a que comenzamos en la siguiente abscisa, descartaríamos aquellas combinaciones que hayan llegado a la última columna con una menor cantidad de breakpoints que K, puesto que el rango del for sería vacío.

En términos de complejidad computacional, su enfoque de fuerza bruta es perjudicial para el tiempo de ejecución en problemas con un gran número de posibles combinaciones, lo cual puede ser solucionado mediante la implementación de podas como veremos a continuación.

2.2 Backtracking

Partiendo del algoritmo anterior, implementamos uno basado en backtracking incorporando una poda por optimalidad y tres por factibilidad con el objetivo de "podar", es decir, eliminar, aquellas ramas que no conduzcan a una solución óptima.

Como poda por optimalidad, decidimos verificar si el error de la combinación parcial actual es menor al mínimo error encontrado. En caso de ser mayor, no tendría sentido seguir explorando dicha rama, ya que podemos asegurar que no llevará a una combinación mejor, puesto que los errores de los puntos no pueden ser negativos. Por otra parte, si la combinación actual no tiene breakpoints o si tiene exactamente uno, decidimos que la función que calcula el error total devuelva cero para seguir explorando esa rama, ya que no podemos descartar que no se pueda extender a una solución óptima.

Luego, para la primera poda de factibilidad, optamos por explorar solamente la primera columna de la discretización si aún no se han agregado breakpoints a la solución actual. De este modo, nos aseguramos de que la función sea factible y evitamos explorar cualquier rama en la que las combinaciones no comiencen en la primera columna.

De manera similar a la poda anterior, si sólo falta un breakpoint para completar la solución, decidimos explorar solamente la última columna para garantizar factibilidad.

Como última poda, decidimos que m1 - (K - len(actual)) sea el límite superior (incluido) del rango de iteración para las abscisas cuando hay al menos un breakpoint en la combinación actual pero aún queda más de un breakpoint por agregar. Realizamos esto para garantizar que los breakpoints restantes (K - len(actual)) tengan espacio suficiente y poder descartar aquellas combinaciones que contengan menos breakpoints de los requeridos.

Finalmente, con el objetivo de evitar repetir tanto código, declaramos la función recorrerOrdenadas. Como su nombre lo indica, esta función explora todas las ordenadas que se encuentran en

la columna pasada por parámetro.

Al igual que el algoritmo de fuerza bruta, también tiene complejidad exponencial, pero resulta en un rendimiento considerablemente mejor debido a las podas utilizadas, puesto que cumplimos el objetivo de evitar explorar ramas que no conduzcan a una combinación óptima.

2.3 Programación dinámica Top-Down

Para implementar el algoritmo con la técnica de programación dinámica Top-Down, utilizamos la estructura recursiva del problema para guardar los resultados de los subproblemas en una matriz tridimensional de memoización. Esta técnica sirve para evitar la repetición de cálculos innecesarios, al utilizar los resultados almacenados en la matriz.

En cada llamada recursiva, verificamos si el valor del mínimo error para los parámetros (M, i, j) ya ha sido calculado y almacenado en la matriz de memoización. Sí está guardado, lo retornamos; si no, comenzamos con el cálculo.

El paso recursivo se ejecutará siempre que quede más de una pieza por colocar. Aquí, calculamos recursivamente el mínimo error usando M piezas cuando el último breakpoint es el recibido por parámetro. Para ello, comenzamos a recorrer todas las posiciones de la grilla a partir de la abscisa M - 1 con el objetivo de considerar el espacio suficiente para las piezas anteriores e iteramos hasta la abscisa anterior al último breakpoint. Luego, para cada una de estas posiciones, calculamos el error de la pieza determinada por el punto actual y el último breakpoint, y sumamos este error al costo mínimo obtenido recursivamente para M - 1 piezas cuando el último breakpoint es el punto actual.

Por otro lado, en el caso de que M sea igual a 1, es decir, que nos queda sólo una pieza por colocar, entramos al caso base. En el mismo, calculamos el mínimo error para todas las funciones de una pieza que tengan el primer breakpoint en la primera columna de la discretización. Este cálculo se realiza recorriendo todas las posibles posiciones de la grilla de discretización, calculando el error de la pieza determinada por cada uno de dichos puntos y algún punto en la primera columna de la grilla de discretización.

Luego, para calcular el error total de solución óptima, llamamos a la función para cada posición en la última columna de la discretización y buscamos el mínimo de los valores devueltos por cada uno de dichos llamados.

Finalmente, teniendo el cubo con los valores necesarios, reconstruimos la solución. Para ello, buscamos el índice del último breakpoint recorriendo la última columna y lo agregamos a la solución. Luego, mientras haya piezas por colocar, recorremos las posiciones que se encuentren en columnas

anteriores a la del último breakpoint agregado a la solución. Dada una posición, vemos cuál es la diferencia entre el valor guardado en el cubo para dicha posición con una pieza menos y el valor correspondiente al último breakpoint. Si esa diferencia es igual al error cometido por la pieza definida por esos puntos, significa que el primer punto se encuentra en la combinación óptima, por lo que dejamos de iterar mediante el uso de una flag para agregar el punto encontrado y seguir buscando con una pieza menos. En el caso de que quede una pieza por colocar, buscamos el punto que se encuentre en la primera columna que haga que el valor en la matriz correspondiente al último breakpoint agregado con una pieza sea igual al error de la pieza entre ellos. Una vez colocadas todas las piezas, guardamos la combinación de manera inversa en el diccionario que contiene la mejor solución encontrada.

2.4 Programación dinámica Bottom-Up

Al haber finalizado el algoritmo de Top-Down, decidimos implementar el algoritmo de programación dinámica con un enfoque Bottom-Up. En este mismo, primero resolvemos los subproblemas que tienen una menor cantidad de piezas y guardamos los resultados en una matriz tridimensional, realizando tableau-filling.

No obstante, a diferencia de otros algoritmos Bottom-Up que llenan toda la "tabla", esta implementación es más selectiva y no calcula ciertas soluciones que podemos asegurar que son innecesarias, similar a la tercera poda de factibilidad en backtracking.

Primeramente, calculamos el error de los subproblemas con una sola pieza y llenamos el cubo en la celda correspondiente. Además, garantizamos el espacio necesario para las piezas restantes (M - 1) con que el límite superior del rango sea m1 - (M -1), evitando así hacer cálculos innecesarios.

Una vez hecho eso, recorremos los subproblemas restantes con una mayor cantidad de piezas. Luego, iteramos sobre las abscisas posibles, comenzando por m para asegurar el espacio de las m - 1 piezas anteriores y terminando en m1 - (M - m) (sin incluir), para el espacio de las restantes (M - m) y para evitar llenar valores que serían parte de combinaciones con menos piezas de las requeridas. Luego, para calcular el error mínimo, sólo recorremos las abscisas que se encuentran antes de p2. Así, sumamos el error de la pieza actual con el mínimo error cometido de todas las funciones PWL de m - 1 piezas que se "peguen" con la pieza actual.

Una vez que tenemos el cubo con los valores necesarios, buscamos el mínimo error y reconstruimos la solución de la misma manera que en Top-Down.

2.5 PD: Top-Down refinada - Implementación extra

Al analizar los algoritmos implementados, notamos que calculamos varias veces el error cometido por la aproximación de una pieza de manera innecesaria. También, observamos que la cantidad de dichos cálculos innecesarios aumentaba a medida que m1 y m2 incrementaban.

Con el objetivo de solucionar este problema, decidimos aplicar los conocimientos de la materia e implementar un grafo pesado, cuyos nodos representan las posibles posiciones que pueden tomar los breakpoints en la grilla y cuyas aristas representan el error cometido por la pieza definida por los puntos correspondientes a los vértices.

Como estructura para representar el grafo, optamos por una matriz de adyacencia, en donde el elemento en la posición (x, y) haga referencia al error cometido por la pieza definida por los vértices x e y. En caso de que no exista una arista, el valor es -1. Para cada una de las posiciones en la grilla de discretización, necesitamos el error de todas las piezas no verticales que contengan a dicha posición. De este modo, al ser un grafo denso, una matriz de adyacencia permite un acceso más rápido a los pesos de las aristas que una lista de vecinos o aristas.

Al inicializar el grafo, construimos una matriz llena de -1 de tamaño n x n, donde n es igual a m1 * m2. De esta manera, los vértices son enumerados de 0 a n - 1. Para enumerarlos, recorremos las posiciones de la grilla de abajo a arriba, y luego de izquierda a derecha, tal como muestra la siguiente representación:

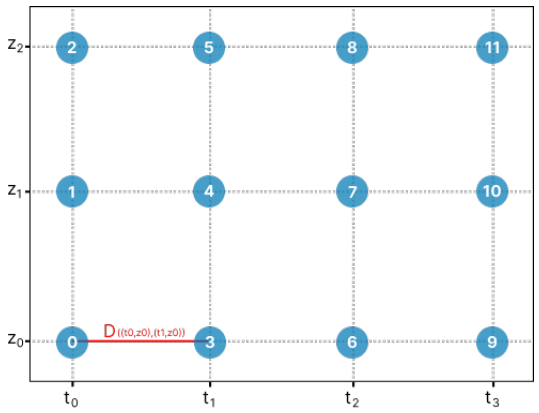

Figure 1: Representación del grafo utilizado

Luego, dada una tupla de índices (i, j) de un breakpoint, podemos obtener el número del vértice correspondiente a dicho punto multiplicando i por m2 y sumándole j. Por ejemplo, el breakpoint (0, 0) correspondería al vértice 0, mientras que el breakpoint (m1 - 1, m2 - 1) correspondería al vértice n - 1.

Debido a que el algoritmo Top-Down es el que, generalmente, tiene un menor tiempo de eje-

cución, decidimos agregar el uso del grafo a esta implementación para observar si podíamos lograr un tiempo de ejecución aún menor.

En primer lugar, al momento de encontrar el error de la mejor combinación, verificamos si el error cometido por una pieza ya fue calculado, es decir, si existe la arista entre los dos extremos de la pieza. De ser así, devolvemos el peso de la arista. De lo contrario, calculamos el error de la pieza y agregamos la arista al grafo para evitar hacer el cálculo nuevamente, similar a lo que realizamos con la matriz de memoización.

Finalmente, para reconstruir la solución, ya contamos con todas las aristas necesarias para obtener la combinación de breakpoints. Por lo tanto, dadas dos tuplas de índices de posibles breakpoints, devolvemos directamente el peso de la arista, disminuyendo la cantidad de cálculos realizados.

3 Consideraciones generales

Al momento de implementar los distintos algoritmos, observamos que compartían algunas variables que nunca eran modificadas, tales como los datos del archivo JSON, la cantidad de breakpoints, la cantidad de filas y columnas, y la grilla de la discretización. Por lo tanto, decidimos que dichas variables sean globales para evitar tener que pasarlas por parámetro a cada función que las necesite, mejorando la organización y claridad del código.

En segundo lugar, al implementar la matriz tridimensional de memorización, optamos por incrementar en uno la variable que representa la cantidad de piezas con el objetivo de que los índices de la matriz coincidan directamente con la cantidad de piezas, lo cual facilita el acceso y la comprensión del código. Por ejemplo, cuando se necesita acceder a algún valor de la matriz con tres piezas, utilizamos el índice tres en lugar de dos, lo que resulta más intuitivo. No obstante, debido a que no hay funciones continuas PWL con cero piezas, esta decisión conlleva a que las primeras posiciones de la matriz queden vacías, lo cual puede ser considerado un desperdicio de memoria. A pesar de esto, decidimos utilizar esta estrategia por comodidad en el manejo de los índices del cubo.

Por otra parte, notamos que para un conjunto de datos, podía haber más de una combinación óptima para ciertos parámetros K, m1 y m2. Debido a esto, dependiendo del algoritmo utilizado para encontrar la mejor aproximación, obteníamos distintos resultados. Al analizar mejor las implementaciones, concluimos que dichas diferencias se deben a la forma de recorrer las distintas posiciones de la discretización. Mientras que los algoritmos de fuerza bruta y backtracking recorren de izquierda a derecha, las implementaciones de programación dinámica reconstruyen la solución

de manera inversa, lo cual conlleva a resultados diferentes ya que los algoritmos pueden encontrar distintas combinaciones de piezas que cometan el mismo error.

Por último, la diferencia entre C++ y Python de cómo manejan la precisión de números de punto flotante presentaron ciertas dificultades al momento de reconstruir la solución en programación dinámica. Al comparar los valores guardados en la matriz tridimensional para decidir si agregar un breakpoint a la solución, notamos que C++ generaba problemas y determinaba que dos valores eran distintos cuando Python los consideraba iguales. Para solucionar este problema, en C++ decidimos utilizar una pequeña tolerancia (EPSILON) para determinar si dos números son "casi iguales" en lugar de exactamente iguales con el objetivo de evitar problemas de precisión.

4 Experimentación

4.1 Comparación de los algoritmos - Python

Al experimentar con nuestras implementaciones, decidimos comenzar analizando diferentes valores de m1, m2 y K, con el objetivo de observar si existen variaciones significativas en los tiempos de ejecución.

Específicamente, esperamos que la programación dinámica Bottom-Up (BU) y Top-Down (TD) demuestren una mayor eficiencia en comparación con los enfoques de fuerza bruta y backtracking para valores más grandes. Esto se debe a las características de la técnica de programación dinámica, que evita cálculos innecesarios y optimiza la resolución de subproblemas.

Para analizar el desempeño de los algoritmos bajo condiciones de gran escala, seleccionamos los valores **m1=30, m2=30 y K=10**. Nuestro objetivo principal es comprender cómo afectan estos parámetros al tiempo de ejecución de cada algoritmo y determinar si alguno de ellos es más eficiente que los demás.

Algoritmo	**Toy Instance**	**Aspen Simulation**	**Ethanol Water Vle**	**Titanium**	**Optimistic**
Fuerza Bruta	> 600	> 600	> 600	> 600	> 600
Backtracking	7.390198231	> 600	> 600	> 600	> 600
PD: top-down	4.158182621	10.48290396	15.19594598	59.37974811	241.7198114
PD: bottom-up	4.640242577	12.45008349	18.56976628	72.26970339	262.3521504
Error total	0	1.8519136	0.06793850575	0.7228821839	2995.495645

Table 1: Resultados para m1=30, m2=30 y K=10

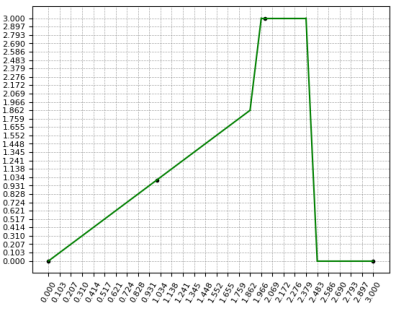

Figure 2: Toy Instance

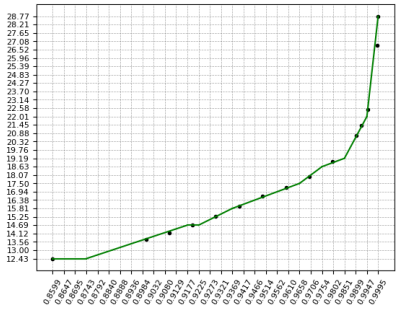

Figure 3: Aspen Simulation

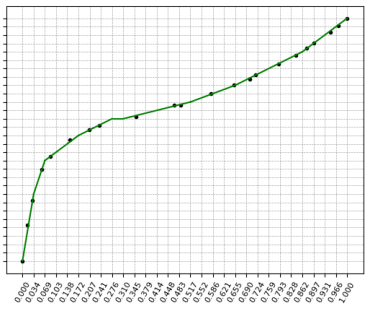

Figure 4: Ethanol Water Vle

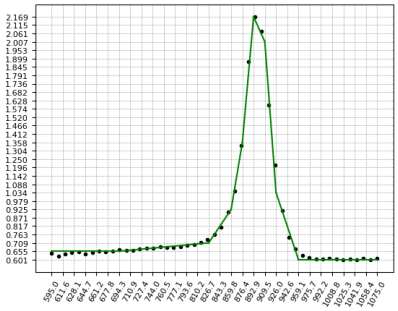

Figure 5: Titanium

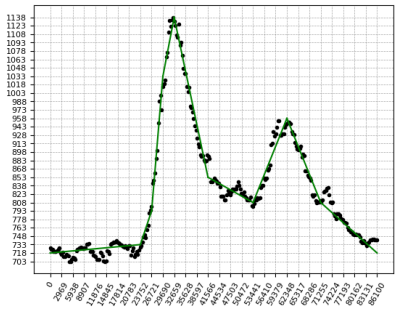

Figure 6: Optimistic

Como podemos ver, efectivamente se cumple nuestra hipótesis. Al comparar los algoritmos entre sí para una misma instancia, encontramos que los algoritmos de fuerza bruta y backtracking tienen un mayor tiempo de ejecución debido a su complejidad exponencial, mientras que los tiempos de ejecución para programación dinámica disminuyen drásticamente. Dada la considerable cantidad de posibles combinaciones de breakpoints, los primeros algoritmos sufrían un aumento significativo en el tiempo de ejecución, por lo que establecimos un límite de 600 segundos para evitar largos períodos de espera sin comprometer la integridad del análisis.

Además, para un mismo algoritmo, notamos que el tiempo de ejecución aumenta a medida que la cantidad de puntos en la instancia de datos incrementa, lo cual se debe mayormente a la complejidad lineal que presentan las funciones que calculan los errores de las aproximaciones.

Por otro lado, también observamos la diferencia entre los distintos algoritmos de programación dinámica, en donde el enfoque bottom-up cuenta con un tiempo de ejecución ligeramente mayor. A pesar de nuestra implementación selectiva de bottom-up, hay algunos valores calculados que podrían resultar innecesarios, mientras que top-down calcula los valores a medida que los necesita. Dicha diferencia en el llenado de la matriz podría ser la principal razón de la diferencia encontrada.

Luego, para analizar el desempeño de los algoritmos bajo parámetros de tamaño intermedio,

elegimos los valores **m1=10, m2=10 y K=5**. Con ello, buscamos comprender cómo se comportan los algoritmos en situaciones moderadas, identificando tendencias no evidentes en casos extremos.

Algoritmo	**Toy Instance**	**Aspen Simulation**	**Ethanol Water Vle**	**Titanium**	**Optimistic**
Fuerza Bruta	98.67914867	237.7338595	310.595418	> 600	> 600
Backtracking	0.01529741287	3.01731348	0.3394033909	6.648172855	67.62234688
PD: top-down	0.01818943024	0.05065870285	0.07384157181	0.201584816	0.9534580708
PD: bottom-up	0.03101658821	0.05307841301	0.0918571949	0.2633152008	1.138291121
Error total	0	8.560527551	0.3839888889	4.650666667	12912.45993

Table 2: Resultados para m1=10, m2=10 y K=5

Con parámetros intermedios, a pesar de que la relación entre los distintos algoritmos se mantiene, podemos ver una disminución considerable en los tiempos de ejecución. También, podemos observar que backtracking se acerca a programación dinámica en términos de tiempo de ejecución para instancias con una menor cantidad de puntos, lo que demuestra la eficiencia de las podas implementadas. No obstante, fuerza bruta sigue presentando una diferencia notable en comparación con los demás algoritmos.

Finalmente, para analizar el desempeño de los algoritmos bajo parámetros de menor tamaño, elegimos los valores **m1= 5, m2=5 y K=3.** Esperamos que la diferencia entre los tiempos de ejecución de los algoritmos sea mínima puesto que todos deberían ser capaces de manejar valores bajos eficientemente.

Algoritmo	**Toy Instance**	**Aspen Simulation**	**Ethanol Water Vle**	**Titanium**	**Optimistic**
Fuerza Bruta	0	0.01803636551	0.01801681519	0.04377102852	0.1785335541
Backtracking	0	0.01523351669	0	0.01567077637	0.07657289505
PD: top-down	0	0.0006635189056	0	0	0.03397512436
PD: bottom-up	0	0	0	0.005705118179	0.05006313324
Error total	0.6666666667	14.7856185	1.243	9.294666667	15386.50523

Table 3: Resultados para m1= 5, m2=5 y K=3

Efectivamente, encontramos que la diferencia es mínima entre los distintos algoritmos. En estos casos con parámetros de menor tamaño, podemos concluir que la complejidad algorítmica de los algoritmos puede no ser un factor determinante.

4.2 Comparación entre Python y C++

Por otro lado, al implementar las funciones en C++, esperamos que la relación entre los algoritmos se mantenga, pero que el tiempo de ejecución sea menor en comparación con los de Python.

Esto se debe a que C++ es un lenguaje de programación compilado, mientras que Python es un lenguaje de programación interpretado, por lo que C++ debería tener un mejor rendimiento en términos de velocidad de ejecución ya que el código es optimizado antes de su ejecución.

	Toy Instance		Aspen Simulation		Ethanol Water Vle		Titanium		Optimistic	
	Python	**C++**	**Python**	**C++**	**Python**	**C++**	**Python**	**C++**	**Python**	**C++**
Fuerza Bruta	98.67914867	87.8037	237.7338595	230.354	310.595418	421.033	> 600	> 600	> 600	> 600
Backtracking	0.01529741287	0.012014	3.01731348	1.28289	0.3394033909	0.407452	6.648172855	6.77421	67.62234688	80.8745
PD: top-down	0.01818943024	0.024029	0.05065870285	0.045972	0.07384157181	0.091329	0.201584816	0.222178	0.9534580708	1.25294
PD: bottom-up	0.03101658821	0.02755	0.05307841301	0.073522	0.0918571949	0.122845	0.2633152008	0.282348	1.138291121	1.60424
Error total	0	0	8.560527551	8.560527551	0.3839888889	0.383989	4.650666667	4.65067	12912.45993	12912.5

Table 4: Comparación de Python y C++ con m1=10, m2=10 y K=5

Los resultados fueron inesperados ya que Python mostró una eficiencia superior en los tiempos de ejecución de algunas instancias en comparación con C++. Creemos que esto se puede deber a que algunas operaciones pueden ser más eficientes en Python por el uso de bibliotecas optimizadas como NumPy. También es posible que factores más específicos, como la elección del compilador de C++ y la versión del intérprete de Python, hayan contribuido a esta particularidad de los resultados.

4.3 Análisis de promedios

Para comprender mejor las ventajas y desventajas de los algoritmos y para compararlos mejor entre ellos, decidimos calcular un promedio que permita ver la relación entre los diferentes algoritmos de forma más clara. Esto permite obtener una visión más completa del rendimiento de cada algoritmo en diferentes situaciones y facilita la comparación de la eficiencia general entre los mismos.

	Fuerza Bruta	**Backtracking**	**PD: top-down**	**PD: top-down refinada**	**PD: bottom-up**
Tiempo de ejecución promedio	323.1511189	165.6760139	22.15126422	6.451990096	24.79435153

Calculamos el tiempo de ejecución de cada algoritmo utilizando las combinaciones 'm1=5 m2=5 K=3', 'm1=10 m2=10 K=5' y 'm1=30 m2=30 K=10'. Luego, realizamos el promedio general para cada algoritmo utilizando estas combinaciones en todas las instancias evaluadas. Es importante destacar que este promedio no refleja necesariamente el tiempo de ejecución promedio del algoritmo en sí, ya que este puede variar significativamente según la combinación de m1, m2 y K, así como el tamaño de la instancia. Sin embargo, al utilizar las mismas combinaciones e instancias para todos los algoritmos, mantenemos la comparabilidad entre ellos. Los resultados muestran una tendencia consistente, con el tiempo de ejecución más alto correspondiente al algoritmo de fuerza bruta y el tiempo más bajo al de programación dinámica refinada.

4.4 Eficiencia del algoritmo PD: Top-down refinado

Luego de haber analizado los tiempos de ejecución variando los parámetros, el algoritmo y las instancias, decidimos experimentar con la diferencia en el tiempo de ejecución entre el algoritmo que implementa programación dinámica Top-Down y la versión refinada del mismo. El objetivo de esta experimentación será poder observar la eficiencia del uso del grafo al evitar cálculos repetitivos. Para ello, a partir de tres combinaciones de K, M1, M2 de distintos tamaños, observamos cómo se diferencian los tiempos de ejecución.

		5,10,10	10,30,30	10,40,40
Toy Instance	**PD TD**	0.01818943024	4.158182621	16.25992727
	PD TD REF	0.0328502655	3.429018259	12.55633211
Aspen Simulation	**PD TD**	0.05065870285	10.48290396	40.21954298
	PD TD REF	0.04171061516	5.087937593	18.56592345
Ethanol Water Vle	**PD TD**	0.07384157181	15.19594598	60.30421638
	PD TD REF	0.06415343285	6.464825392	24.21358252
Titanium	**PD TD**	0.201584816	59.37974811	265.8523998
	PD TD REF	0.1687166691	16.35125947	65.30889177
Optimistic	**PD TD**	0.9534580708	241.7198114	1120.12966
	PD TD REF	0.7083024979	64.39654636	267.3241515

Table 5: Comparación entre el algoritmo PD: Top-down original y el refinado

Después de observar los resultados de la experimentación, se evidencia que el algoritmo refinado presenta un rendimiento superior en términos de tiempos de ejecución en comparación con la implementación estándar de programación dinámica Top-Down. Esta mejora es especialmente notable en instancias con parámetros más grandes. Por ejemplo, en la instancia Optimistic con K=10, M1=40 y M2=40, el algoritmo de programación dinámica top-down se ejecuta en 1120.13 segundos, mientras que la versión refinada lo hace en 267.32 segundos, lo que representa una diferencia significativa.

Esta tendencia se observa consistentemente a lo largo de las distintas combinaciones de parámetros evaluadas. A medida que aumentan los valores de K, M1 y M2, la diferencia en los tiempos de ejecución entre los algoritmos estándar y refinado se vuelve más pronunciada, lo que sugiere que el aprovechamiento del grafo para evitar cálculos repetitivos tiene un impacto positivo en la eficiencia del algoritmo.

4.5 Regresión lineal

Trabajar con la función PWL nos llevó a relacionarla con el error cuadrático medio (ECM) de la regresión lineal. A pesar de que la función PWL y la regresión lineal utilizan métodos diferentes para minimizar sus respectivos errores, ambos buscan la mejor aproximación a los datos dados. La función PWL lo hace minimizando la suma de las diferencias absolutas entre los valores observados, mientras que la regresión lineal busca minimizar la suma de las diferencias al cuadrado dividida por el número de puntos.

De este modo, implementamos una función auxiliar para calcular la recta de regresión lineal. Sin embargo, dada esa recta, no calculamos el ECM, sino que calculamos la suma de los errores de los puntos al igual que en las funciones PWL. Luego, la comparamos con una función PWL de una pieza (K = 2) con los parámetros m1 = 2 y m2 = 200 para simular continuidad y que los breakpoints tengan "mayor libertad" al tener una cantidad considerable de posibles posiciones.

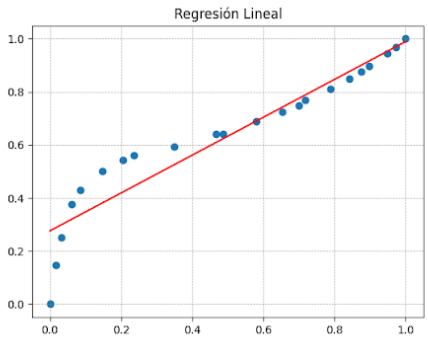

Figure 7: Función de Regresión Lineal, Error total = 1.269611

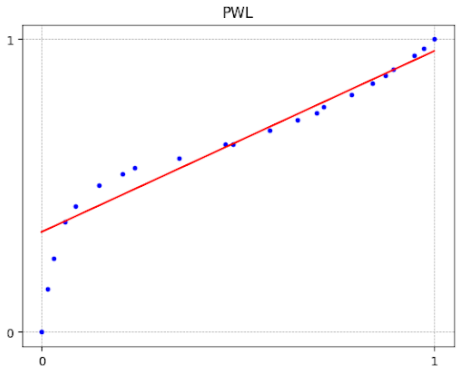

Figure 8: Función PWL, Error total = 1.149913

En todos los casos, observamos que el error cometido por la función PWL de una pieza es menor que el correspondiente a la recta de regresión lineal. Esto era de esperarse, ya que PWL está diseñada específicamente para minimizar dicho error. Sin embargo, notamos que, en general, la diferencia entre ambos errores no es significativa en el contexto de las distintas instancias.

4.6 Casos atípicos

Al experimentar con distintos parámetros, nos llamó la atención el comportamiento de algunas combinaciones óptimas. En particular, cuando los valores de los parámetros eran mayores a la cantidad de puntos de la instancia y la diferencia entre K y m1 era mínima, observamos que los gráficos de las funciones mostraban picos pronunciados, tal como muestra la siguiente imagen:

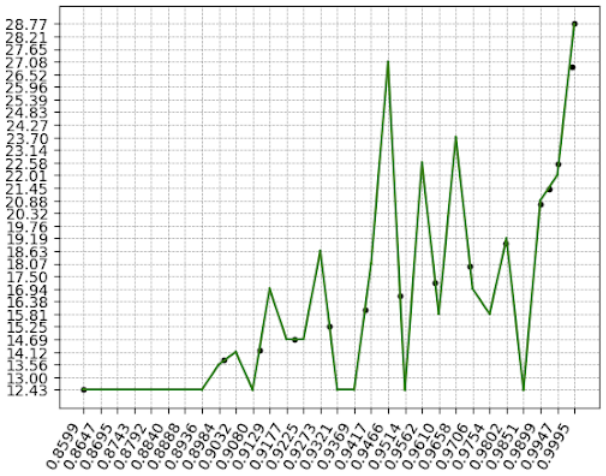

Figure 9: aspen simulation con K = m1 = m2 = 30

Si bien no conocemos la función que intentamos aproximar, al observar la tendencia que presentan los puntos de las instancias, notamos que los picos no parecen estar alineados con la forma esperada de la función desconocida. Creemos que este comportamiento se debe a la gran libertad que presenta una discretización tan extensa, lo cual permite que las distintas piezas puedan ignorar las aproximaciones de las demás piezas para lograr minimizar el error.

5 Conclusión

A lo largo de este trabajo, hemos analizado en profundidad los distintos algoritmos desarrollados para abordar el problema de la aproximación de datos vía funciones lineales continuas a trozos. Nuestro objetivo principal fue analizar y comparar el rendimiento de diferentes enfoques al resolver este problema, evaluando aspectos como el tiempo de ejecución, la precisión en la aproximación y la escalabilidad para instancias de diferentes tamaños. En base a nuestro análisis, pudimos llegar a diversas conclusiones al respecto.

En primer lugar, sostenemos que el algoritmo más óptimo para utilizar ante este problema sería el de programación dinámica. El mismo, demostró ser más eficiente al adaptarse al problema, resolviéndolo en menos tiempo. Esto lo pudimos notar especialmente para valores de m1, m2 y K de mayor tamaño. Además, al introducir el refinamiento en la implementación del algoritmo de programación dinámica, con el uso de grafos para evitar cálculos redundantes, logramos mejorar aún más su eficiencia y escalabilidad.

Por otro lado, respecto al lenguaje de programación, Python nos pareció el más efectivo ya que,

en general, los tiempos de ejecución terminaron siendo más rápidos. Sin embargo, para ciertos valores de los parámetros o para instancias específicamente de baja escala, creemos que puede ser beneficioso considerar el uso de C++.

Por último, hemos observado ciertos patrones consistentes que emergieron a través de la experimentación con parámetros óptimos. Por ejemplo, en conjuntos de datos con una alta densidad de puntos, notamos que valores más elevados de m1 y m2, junto con un valor de K moderado, tendían a producir mejores resultados en términos de precisión en la aproximación. Sin embargo, es importante tener en cuenta que siempre se debe ajustar los parámetros según las características específicas del problema en cuestión.

En base a lo mencionado, creemos que nuestro trabajo no sólo nos permitió comprender cómo difiere el rendimiento de diferentes estrategias algorítmicas, sino que también nos proporcionó una comprensión más profunda de cómo mejorar tanto la precisión como la eficiencia en la aproximación de funciones utilizando PWL.