UNIVERSIDAD TORCUATO DI TELLA

TP1 - Aproximación de datos vía funciones lineales continuas a trozos

Tecnología Digital V: Diseño de Algoritmos - Prof. Juan Jose Miranda Bront

1er semestre 2024

CHAB LÓPEZ, Catalina

CHEN, Belén

FERREIRO GRAU, Luna

1. Introducción

La aproximación de datos mediante funciones continuas es una herramienta indispensable en numerosas disciplinas, abarcando desde la ingeniería hasta la economía, permitiendo modelar eficazmente fenómenos complejos con diversos comportamientos. En este contexto, las funciones lineales continuas a trozos (PWL, por sus siglas en inglés) emergen como una herramienta útil. La naturaleza de muchos fenómenos complejos, cuyas funciones subyacentes son desconocidas, exige estrategias para aproximarlas de manera precisa.
En este trabajo, nos enfocaremos en el desarrollo de métodos para aproximar funciones desconocidas mediante funciones continuas PWL, minimizando el error de aproximación. Para ello, diseñaremos y evaluaremos diferentes algoritmos, implementándolos en Python y C++, y experimentando para analizar tanto la calidad como la performance de las aproximaciones obtenidas en diversas instancias.

2. Diseño de los algoritmos

Funciones auxiliares: error

- Descripción:

  La función **error** calcula el error acumulado de un conjunto de puntos **x** e **y** en relación con una línea recta que se define a partir de dos breakpoints dados por **sol**. La función mide qué tan bien se ajusta la línea recta que une ambos breakpoints a los puntos $(x_i, y_i) \in$ **x**, **y** dentro del intervalo definido por la línea recta.

- Implementación:

  En esta función, se inicializa el error (**res**) en cero y se calcula la recta '**m**x + **c**' que une los breakpoints **a** y **b,** obtenidos de los últimos dos elementos de una lista de tuplas **sol**.

Luego, recorre los puntos (x_i, y_i) en el rango definido por los breakpoints y suma el error absoluto entre el valor real **y[i]** y el valor de la recta (m * **x[i]** + c) en ese punto.
Finalmente, devuelve el error total acumulado 'res'.
Decidimos que reciba una lista de tuplas en lugar de dos tuplas por separado porque permite evaluar errores utilizando soluciones parciales generadas por métodos de fuerza bruta y backtracking.

Fuerza bruta

- Descripción:
  El algoritmo de fuerza bruta es una técnica de búsqueda que prueba todas las posibles soluciones de un problema y devuelve la óptima. En el contexto de este trabajo, el algoritmo de Fuerza Bruta busca todas las combinaciones posibles de aproximaciones en una discretización de una grilla **m1**, **m2** para generar todas las funciones continuas PWL candidatas y comparar el error total de cada una de ellas, devolviendo la de error total mínimo.
- Implementación:
  Variables utilizadas
  K: Representa la cantidad de breakpoints. Se asume que siempre es menor o igual a la discretización del eje x de la grilla ($K \leq m1$), garantizando así que nunca reciba una cantidad de breakpoints que no puedan ubicarse. Además, se establece que K siempre es mayor o igual a 2 ($K \geq 2$), que es la cantidad mínima de breakpoints que se pueden colocar (uno al principio y uno al final).
  m1: Cantidad de puntos en la discretización de la abscisa.
  m2: Cantidad de puntos en la discretización de la ordenada.
  N: Indica la cantidad de puntos del conjunto de datos x, y.
  grid_x, grid_y: Son listas que contienen los valores de la discretización en los ejes **x** e **y**, respectivamente.
  x, y: Son listas que contienen las coordenadas de los puntos del conjunto de datos.
  best: Es un diccionario que almacena la mejor solución encontrada hasta el momento. Contiene dos claves: **'sol'**, que representa la lista de tuplas de breakpoints de la mejor solución, y **'error'**, que almacena el error de los breakpoints de la mejor solución.
  current: Es un diccionario que almacena la solución actual en proceso de evaluación. Similar a best, contiene una lista de tuplas de breakpoints (**'sol'**) y el error de los breakpoints (**'error'**).
  i: Es un iterador sobre el eje x de la grilla.

```
best['sol'], current['sol'] <- []
best['error'] <- BIG_NUMBER ; current['error'] <- 0
i=0
FuerzaBruta(K, ... , best, current, i)
```

```
# CASO BASE
if K = 0 or i = |grid_x| entonces
   if K = 0 and current['error'] < best['error'] entonces
       best <- current
# PASO RECURSIVO
else
  para cada valor de grid_y[j] hacer
       if i = 0 entonces # 1er columna => 1er bp
          current['sol'].append((grid_x[0], j))
          FuerzaBruta(K - 1, ..., i + 1)
          current['sol'].pop() # borrar últ bp para probar otras posibles sol
       elif K = 1 o i = |grid_x| - 1 - 1 entonces # último bp
          current['sol'].append((grid_x[-1], j))
          current['error'] += error_parcial()
          FuerzaBruta(K - 1, ..., i + 1)
          current['error'] -= error_parcial()
          current['sol'].pop()
       else
          FuerzaBruta(K, ..., i + 1) # no agrego
          current['sol'].append((grid_x[i], j)) # agrego
          current['error'] += error_parcial()
          FuerzaBruta(K-1, ..., i + 1)
          current['error'] -= error_parcial()
          current['sol'].pop()
```

Backtracking:

- Descripción:

  El algoritmo de backtracking es similar al de Fuerza Bruta, pero incorpora podas por factibilidad y por optimalidad para reducir el espacio de búsqueda y mejorar la eficiencia.

- Implementación:

  Las variables que toma nuestro algoritmo de backtracking son las mismas que toma fuerza bruta, pero agregando una variable **k_0**, que va a guardar la cantidad original de breakpoints a colocar K. Esta variable nos va a permitir imponer nuestra poda por factibilidad, asegurándonos de solo buscar el óptimo entre secuencias con exactamente **K** breakpoints.

  La colocación del primer y el último breakpoint funciona de la misma manera que fuerza bruta. Las podas son implementadas en los pasos intermedios y son las siguientes:

  - **Poda por factibilidad:** Solo se permite agregar un breakpoint si el tamaño restante de la grilla es suficiente para colocar el resto de los breakpoints necesarios para alcanzar el mínimo requerido (**k_0**).
  - **Poda por optimalidad:** Solo vamos a agregar el siguiente breakpoint si el error total que genera agregar dicho breakpoint es menor a nuestro **best['error']** (el menor error encontrado hasta ahora).

```
...
k_0 = K
Backtracking(K, ... , best, current, i, k_0)
  # CASO BASE
```

```
  ...
 # PASO RECURSIVO
 else
   para cada valor de grid_y[j] hacer
      if i = 0 entonces # 1er columna => 1er bp
           ...
      elif K = 1 o i = |grid_x| - 1 entonces # último bp
           ...
      elif current['error'] < best['error'] and |grid_x[i:]| >= k_0 -
|(current['sol'])|:
       if not estoy_en_anteult_col(i):
          Backtracking(K, ..., i + 1) # no agrego
       current['sol'].append((grid_x[i], j)) # agrego
       current['error'] += error_parcial()
       Backtracking(K-1, ..., i + 1)
       current['error'] -= error_parcial()
       current['sol'].pop()
```

Programación Dinámica:

- Descripción:

  Este algoritmo utiliza la técnica de memoización para almacenar y reutilizar los resultados de subproblemas ya resueltos. Se define una estructura que nos permite guardar información relevante al problema que buscamos resolver, a la cual vamos a poder acceder posteriormente para hacer nuestra búsqueda más eficiente.

- Implementación:

  Variables

  En el caso de programación dinámica, no tenemos variables que almacenen resultados parciales o óptimos como en los algoritmos previos, ya que es una función **iterativa** y no **recursiva.**

  Observación: En este algoritmo, **K** hace referencia a la cantidad de **piezas**, no de **breakpoints**. El parámetro es pasado en la llamada a función como **K** (cantidad de breakpoints) **-1**.

  Construcción del memo

  En primer lugar inicializamos nuestro **memo** como una matriz tridimensional de tamaño (K+1)×m1×m2. Inicialmente, se inicializa el memo con **BIG_NUMBER** para representar un error infinito en posiciones no alcanzables con cierta cantidad de piezas, tal como 'memo[0][i][j]', no definido porque no se puede llegar a ningún punto con 0 piezas. Además, siempre que **K > i**, es imposible llegar a **i** con K piezas, por lo que se mantiene **BIG_NUMBER** en esas posiciones. Para la estructura $\mathbf{arr_1[arr_2[arr_3[float]]]}$, $\mathbf{arr_3}$ almacena el error mínimo para llegar a cada coordenada (**i,j**) con cierta cantidad de piezas. Es decir, acceder a **memo**[3][5][4] nos devolvería la manera óptima de llegar al punto (5,4) con exactamente 3 piezas.

  De esta manera, construimos el memo en 2 partes: primero el nivel **K** = 1, y luego, a partir de este el resto de los niveles (siendo que se computan los errores en base a los valores computados en el nivel anterior).

```
dp_2(K, x, y, grid_x, grid_y, m1, m2):
 memo <- matriz tridimensional de K+1 niveles, m1 filas, m2 columnas,
 inicializada con BIG_NUMBER
 # NIVEL 1: Calcular los errores mínimos para 1 pieza
 para cada i desde 1 hasta m1 - 1:
     para cada j desde 0 hasta m2 - 1:
         min_error <- BIG_NUMBER
         para cada p desde 0 hasta m2 - 1:
             error_int <- error((grid_x[0], grid_y[p]), (grid_x[i], grid_y[j]))
             min_error <- mínimo(min_error, error_int)
         memo[1][i][j] <- min_error

 # RESTO DE NIVELES
 para cada k desde 2 hasta K:
     para cada i desde 2 hasta m1 - 1:
         para cada j desde 0 hasta m2 - 1:
             min_error <- BIG_NUMBER
             para cada o desde 0 hasta i - 1:
                para cada p desde 0 hasta m2 - 1:
                   error_int <- error((grid_x[o], grid_y[p]), (grid_x[i],
 grid_y[j])) + memo[k-1][o][p]
                   min_error <- mínimo(min_error, error_int)
             memo[k][i][j] <- min_error
 return memo
```

Reconstrucción

Una vez completo el memo, el siguiente paso es la reconstrucción de la solución óptima. Primero, se elige el último breakpoint, que será posicionado en las coordenadas que ofrezcan el menor error acumulado en la última columna del nivel **K.** Una vez fijado el último breakpoint, retrocedemos en los niveles del memo para encontrar los breakpoints restantes. Para cada nivel **K** buscamos el breakpoint anterior que condujo al mínimo error acumulado para el punto actual. Continuamos este proceso hasta que llegamos al primer nivel (**K**=1), reconstruyendo así todos los breakpoints de la aproximación óptima. Decidimos en principio guardar en nuestra variable de retorno una lista de **posiciones**, no de valores, para facilitar la reconstrucción. Como último paso, se hace la traducción de posiciones a valores de la grilla.

```
dp(K, x, y, m1, m2):
 memo <- dp_2(K, x, y, grid_x, grid_y, m1, m2)
 mejor_solución = [] , mejor_error = BIG_NUMBER

 # Buscar el punto con el error mínimo en la última columna de la grilla.
 para j desde 0 hasta m2 - 1:
     si memo[K][m1 - 1][j] < mejor_error:
         mejor_error = memo[K][m1 - 1][j]
         último_punto_y = j
 mejor_solución.append((m1 - 1, último_punto_y))

 # Reconstruir los puntos clave de los breakpoints previos de derecha a
 izquierda.
 para k desde K - 1 hasta 1:
     mejor_punto = encontrar_mejor_punto(k, m1, m2, memo, mejor_solución)
     mejor_solución.insert(0, mejor_punto)
```

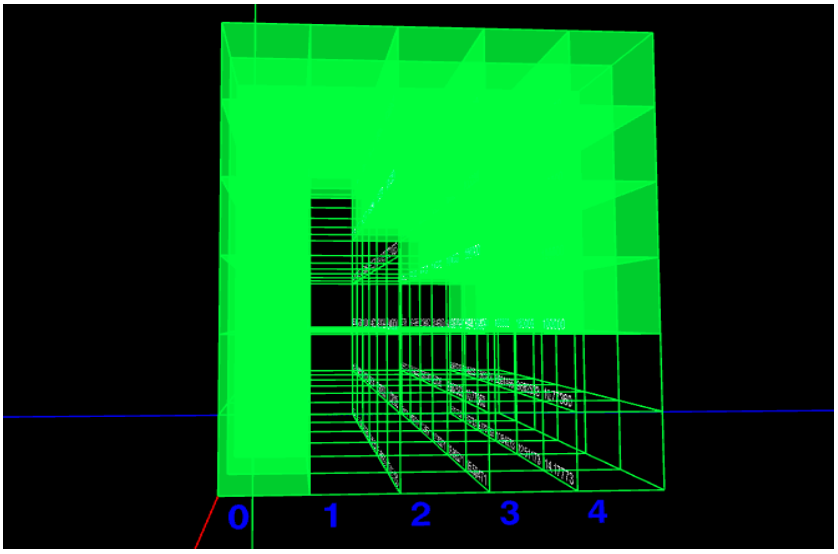
```
# Añadir el primer punto.
primer_punto_y = encontrar_primer_punto_y(m2, mejor_solución, memo)
mejor_solución.insert(0, (0, primer_punto_y))

# Convertir los índices de la solución a puntos de grid_x y grid_y.
para cada índice en mejor_solución:
    agregar_punto_a_solución(mejor_solución, grid_x, grid_y)

# Retornar la mejor solución y error.
return mejor_solución, mejor_error
```

Extra

Usamos un visualizador de matrices tridimensionales[1] como herramienta para ayudarnos a comprender cómo se llenaba el memo, donde quedaban los valores "infinitos" y si estaban en las posiciones correctas.



Ref: Los valores azules son la cantidad de piezas. Podemos ver como con 0 piezas ningún punto es alcanzable, y cómo a medida que van incrementando las piezas otros dejan de serlo también.

Detalles de implementación

- Una diferencia notable entre C++ y Python es la facilidad de crear una grilla de valores equiespaciados en Python usando la función **linspace()** de **NumPy**. En C++ se requiere crear la grilla manualmente, iterando sobre los puntos y añadiéndolos uno por uno.
- Otra gran diferencia entre ambos lenguajes son los tipos de datos. En Python guardamos la solución en un map, donde una de las claves tiene asociado el conjunto de breakpoints y otra el error asociado a ese conjunto de breakpoints. En C++, al ser más estricto con los tipos de datos, cambiamos esa estructura a una tupla que guarda por un lado el vector con el conjunto de breakpoints y por otro el error asociado.

[1] https://array-3d-viz.vercel.app/

3. Experimentación

Evaluación

- Calidad: ¿Cómo impacta la granularidad de la discretización y la cantidad de piezas seleccionadas en la calidad de las aproximaciones?

  A medida que aumenta la granularidad de la discretización se espera una mejora en la precisión de la aproximación. Una discretización más fina permite capturar detalles y variaciones sutiles en la función original. De todas formas, no siempre observamos eso en la experimentación. En el caso de **titanium.json**, notamos que incrementar el tamaño de la grilla no siempre mejora el error. Para una grilla de 6x6 encontramos un error menor que para una grilla de 7x7. Por otro lado, en el caso de **water_ethanol_vle.json**, si notamos lo esperado. Con 3 breakpoints y partiendo de una grilla de 4x4, fue mejorando la aproximación a medida que aumentamos el tamaño de la grilla. Creemos que esto puede deberse a una cualidad intrínseca a los datos de dicha instancia que beneficia a una grilla de un tamaño puntual por sobre las otras.

  Lo mismo sucede con la cantidad de piezas. En principio, aumentar la cantidad de piezas da más flexibilidad y permite un mejor ajuste a los datos. Sin embargo, notamos en la mayoría de las instancias que llega un momento en el que aumentar la cantidad de breakpoints empeoraba la aproximación. Creemos que esto puede deberse a un sobreajuste o overfitting, donde la flexibilidad adquirida por una gran cantidad de breakpoints genera que el modelo capture el ruido en los datos en lugar de la verdadera tendencia subyacente.

  Para evaluar el ajuste de las aproximaciones frente a cambios en los parámetros decidimos observar cómo se modifica el error frente a un cambio en la grilla (**K** estática) y frente a un cambio en **K** (grilla estática).

  Como podemos observar en ambas figuras, no hay una relación estrictamente lineal entre la cantidad de breakpoints o el tamaño de la grilla con la calidad de la aproximación.

  Como podemos ver en la **Figura 1.1**, encontramos casos, como la instancia Aspen, que se benefician de un tamaño mayor de grilla. También observamos instancias que al aumentar el tamaño de la grilla exhiben un error mayor. De todas formas, no podemos asegurar que sucede al seguir incrementando la grilla, e intuimos que eventualmente incrementar mucho la grilla lleva a la posibilidad de poder aproximar de forma muy precisa cualquier set de datos.

  En la Figura 1.2 también podemos observar cómo aumentar la cantidad de breakpoints no tiene una gran influencia en la exactitud de la aproximación, al menos en tamaños de grilla relativamente pequeños como con los que estamos trabajando.

  Figura 1.1: Aumento de grilla - Calidad

  Obs: Todos los valores fueron observados para K=3.

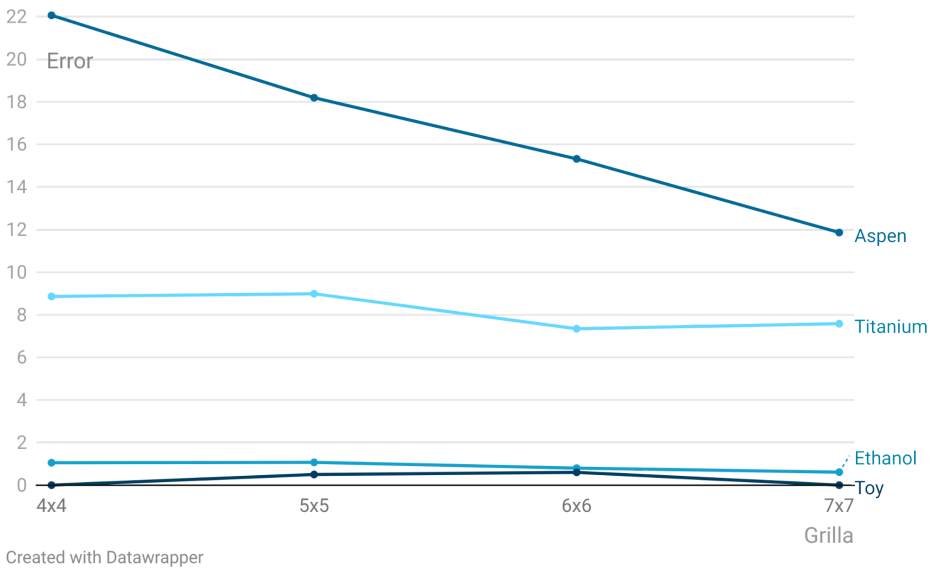
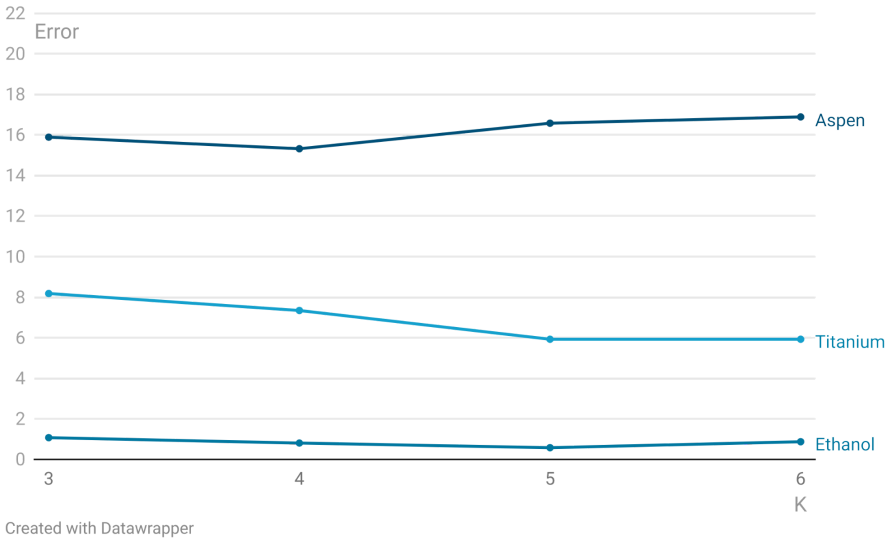


Figura 1.2: Aumento de breakpoints - Calidad

Obs:

- Todos los valores fueron observados para una grilla de 6x6.
- No contemplamos **toy_instance** en la **Figura 1.2** ya que al presentar valores muy pequeños quitaba claridad al gráfico.



Instancia optimistic_instance

Un caso particular que no incluimos en los gráficos anteriores es el del set de datos optimistic_instance. Decidimos analizarlo por separado por 2 razones principales; por un lado, los valores observados con esta instancia son de un tamaño mucho mayor a los observados en las otras instancias, lo que generaba una distorsión en nuestros gráficos. Por otro lado, al ser una instancia con una gran cantidad de puntos, notamos que las aproximaciones se comportan distinto. Como podemos ver en las **Figuras 2.1** y **2.2**, esta instancia es la única que se beneficia estrictamente de aumentar la grilla y de aumentar la cantidad de breakpoints. Lo que concluimos fue que, al tener un tamaño mucho más grande que el resto de las instancias, las grillas que probamos (como mucho de 6x6) y la

cantidad de breakpoints (como mucho 6) no eran suficientes para un set de datos de este tamaño, y cualquier aumento en los parámetros lleva a una mejor aproximación.

Figura 2.1: Aumento de grilla - Calidad (optimistic_instance)

Obs: Todos los valores fueron observados para **K**=3.

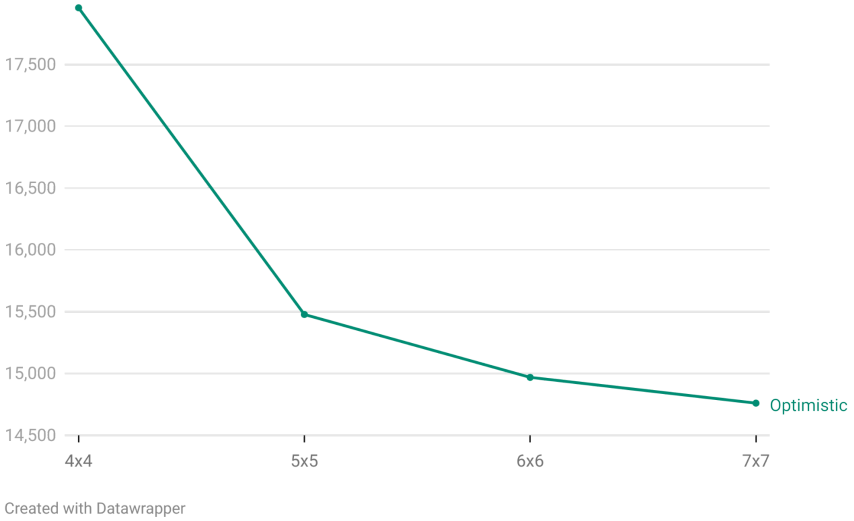
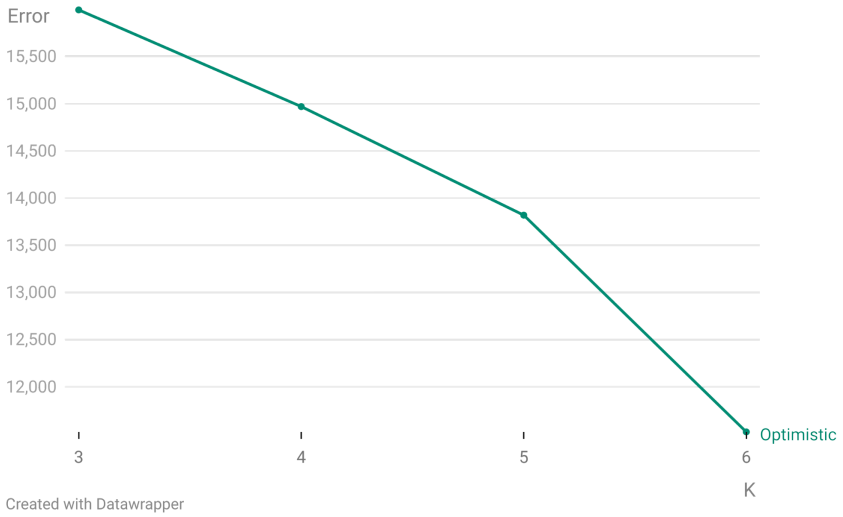


Figura 2.1: Aumento de breakpoints - Calidad (optimistic_instance)

Obs: Todos los valores fueron observados para una grilla de 6x6.



- Performance: ¿Cuáles son los parámetros de las instancias que mayor impacto tienen en la performance de los algoritmos? ¿Cómo afecta la elección del lenguaje de programación y la implementación elegida?

  Para probar qué parámetros son los que más influencian en la performance de los algoritmos decidimos hacer pruebas, por un lado, manteniendo el tamaño de la grilla y modificando la cantidad de breakpoints y, por otro lado, dejando fija la cantidad de breakpoints y modificando el tamaño de la grilla. Para cada algoritmo, la elección de distintos parámetros tiene un impacto diferente en el tiempo de ejecución. En el caso de fuerza bruta, aumentar el tamaño de la grilla o aumentar la cantidad de breakpoints

tiende a generar un aumento en el tiempo de ejecución **(Figuras 3.1 y 3.2)**, aunque notamos que el crecimiento en este tiempo es más veloz al aumentar la grilla que al aumentar los breakpoints. Algo distinto sucede en backtracking. Al considerar sólo las posibilidades factibles, aumentar la cantidad de breakpoints genera una disminución en el tiempo de ejecución, como se observa en la **Figura 3.2.** Esto se debe a que a medida que aumentan los breakpoints, cada pieza tiene menos opciones para ser colocada, ya que impusimos una poda por factibilidad.

Para analizar la performance realizamos una experimentación que se basó en tomar grillas de diversos tamaños y distintas cantidades de breakpoints y corrimos los distintos algoritmos en Python y C++ para analizar el tiempo que tardaban en ejecutarse. Los distintos casos que planteamos para (**K**, **m1**, **m2**) fueron: (5,6,6), (5,7,7), (5,8,8), (2,9,9), (3,9,9). En primer lugar, decidimos fijar 5 breakpoints porque consideramos que es una buena cantidad para aproximar los datos de las instancias provistas, ya que permite tener un buen ajuste de los datos. Así,realizamos pruebas con distintos tamaños de grilla y analizamos el impacto en el tiempo. En segundo lugar, probamos con grillas todavía más grandes (9x9) pero con menos breakpoints permitiendo analizar el impacto que tiene el aumento en un breakpoint en grillas muy grandes.

En torno a las mediciones que tomamos con estos distintos casos, decidimos esperar como máximo alrededor de un minuto al ejecutar cada algoritmo en cada lenguaje. Asimismo, para obtener resultados más precisos, no nos basamos en una sola medición por cada algoritmo en cada lenguaje, sino que ejecutamos cada caso cuatro veces y calculamos el promedio, con el objetivo de contemplar la variabilidad en los tiempos de ejecución. Para agilizar este proceso, creamos una función llamada tiempo_promedio que toma como parámetros la cantidad de breakpoints, las discretizaciones de los ejes, el archivo que se quiere utilizar y el algoritmo con el cual se quieren aproximar los datos y luego devuelve el promedio del tiempo de ejecución de 4 veces la función indicada.

En primer lugar, notamos que el rendimiento varía significativamente dependiendo del lenguaje de programación utilizado. Los lenguajes compilados, como C++, tienden a ser más rápidos porque su código se traduce directamente a instrucciones de máquina específicas antes de la ejecución, mejorando la eficiencia en el tiempo de ejecución, ya que el código se optimiza para el hardware en el que se ejecuta. En cambio, Python es un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea a través de un intérprete en tiempo real, lo que puede impactar negativamente en el rendimiento.

En segundo lugar, se puede ver que para todas las instancias, en general el algoritmo que más demora es fuerza bruta, luego le sigue backtracking y por último programación dinámica. Estos resultados son congruentes con la intuición; uno espera que fuerza bruta sea la implementación menos eficiente, ya que debe probar todas las opciones posibles. Luego, esperamos que backtracking, al podar ciertas opciones que no son factibles u óptimas, tarde menos. Y por último, se espera que programación dinámica sea el que menos tarda, ya que se realizan muchos menos cálculos y se construye la solución una

única vez, en base a los errores pre-computados. Sin embargo, estos resultados se invierten para casos donde hay pocas opciones posibles y las discretizaciones son grandes, como es el caso de (2,9,9). Si solo hay dos breakpoints, estos deben ponerse en la primera y última columna de la discretización del eje x. Lo único que puede variar son los valores de y, por ende las opciones posibles de caminos son pocas. Lo que observamos en este ejemplo es que fuerza bruta y backtracking demoran menos que programación dinámica. Creemos que esto se debe a que como es una grilla grande la que se crea, en programación dinámica creamos un memo demasiado grande en comparación con las posibilidades que existen de soluciones y perdemos tiempo en la construcción de este.

En términos generales, una mayor cantidad de datos suele traducirse en una mayor duración en la ejecución de los algoritmos, ya que la función de error debe realizar más cálculos. No obstante, hay excepciones en las que los resultados no coinciden con lo esperado. Por ejemplo, al comparar las instancias "Aspen" (n = 15) y "Ethanol" (n = 22) en una grilla de 8x8 con k = 5 con backtracking, se observa que "Aspen" tardó más tiempo en ejecutarse que "Ethanol" (**Tablas 2, 3**), lo cual resulta anti intuitivo. Esto podría deberse a varios factores, como detalles de la implementación del algoritmo o la complejidad de los datos: aunque "Ethanol" tiene un tamaño de datos mayor, su estructura interna podría ser más sencilla, permitiendo al algoritmo procesarlos más rápido.

Por último, un caso interesante para analizar es (3,9,9), que nos muestra cómo agregando un breakpoint en una grilla grande las posibles soluciones aumentan en gran cantidad y el tiempo de ejecución supera el minuto. No sabemos precisamente en cuanto, pero lo importante es que aumenta muchísimo en comparación con (2,9,9).

Tabla 1. toy_instance.json *(n = 4)*			Un. de medida: milisegundos		
ALGORITMO \ (K, m1, m2)	**(5, 6, 6)**	**(5, 7, 7)**	**(5,8,8)**	**(2,9,9)**	**(3,9,9)**
FB python	1668,00745	≥ 1 min	≥ 1 min	0,3559	≥ 1 min
FB cpp	607	13958,5	≥ 1 min	0	≥ 1 min
BT python	19,458325	6,648025	46,4884	0,352175	≥ 1 min
BT cpp	4	6,25	42	0	≥ 1 min
DP python	5,5841	18,64305	18,6798	1,704175	10,047275
DP cpp	0,75	2,5	5	0	2
Tabla 2. aspen_simulation.json *(n = 15)*					
ALGORITMO \ (K, m1, m2)	**(5, 6, 6)**	**(5, 7, 7)**	**(5,8,8)**	**(2,9,9)**	**(3,9,9)**
FB python	4277,868275	≥ 1 min	≥ 1 min	0,771325	≥ 1 min
FB cpp	597,75	15902	≥ 1 min	0	≥ 1 min
BT python	215,137352	4171,50285	≥ 1 min	0,709725	≥ 1 min
BT cpp	119,75	1600,25	44411,2	0	≥ 1 min
DP python	12,80255	21,303125	35,5233	3,273525	24,705775
DP cpp	1	2,5	6,25	0	3,25

Tabla 3. ethanol_water_vle.json *(n = 22)*					
ALGORITMO \ (K, m1, m2)	**(5, 6, 6)**	**(5, 7, 7)**	**(5,8,8)**	**(2,9,9)**	**(3,9,9)**
FB python	5838,0286	≥ 1 min	≥ 1 min	1,257	≥ 1 min
FB cpp	647,75	17130	≥ 1 min	0	≥ 1 min
BT python	46,0666	699,5036	11850,3849	1,14332	≥ 1 min
BT cpp	45	329,75	5602	0	≥ 1 min
DP python	17,565	26,677025	48,4855	7,4349	33,494
DP cpp	2	3,75	7	0	2,75
Tabla 4. titanium.json (n = 49)					
ALGORITMO \ (K, m1, m2)	**(5, 6, 6)**	**(5, 7, 7)**	**(5,8,8)**	**(2,9,9)**	**(3,9,9)**
FB python	15.447,2564	≥ 1 min	≥ 1 min	3	≥ 1 min
FB cpp	959,5	22847	≥ 1 min	0	≥ 1 min
BT python	333,1986	7169,573375	≥ 1 min	20,844425	≥ 1 min
BT cpp	98,2500	1367	43.425	0	≥ 1 min
DP python	80,1238	109,46265	221,629925	30,3666	90,520325
DP cpp	3	5,5	9,5	0	3,75
Tabla 5. optimistic_instance.json (n = 288)					
ALGORITMO \ (K, m1, m2)	**(5, 6, 6)**	**(5, 7, 7)**	**(5,8,8)**	**(2,9,9)**	**(3,9,9)**
FB python	65.483,8366	≥ 1 min	≥ 1 min	15,0475	≥ 1 min
FB cpp	2483,75	65964	≥ 1 min	0	≥ 1 min
BT python	1327,8366	46009,014	≥ 1 min	14,9941	≥ 1 min
BT cpp	242	5912,5	≥ 1 min	0,5	≥ 1 min
DP python	185,8767	328,9585	573,8546	70,792	336,433
DP cpp	10,25	16,25	16	3	23,25

Figura 3.1: Aumento de grilla - Performance

Obs: Todos los valores fueron observados para K=3, instancia Titanium.

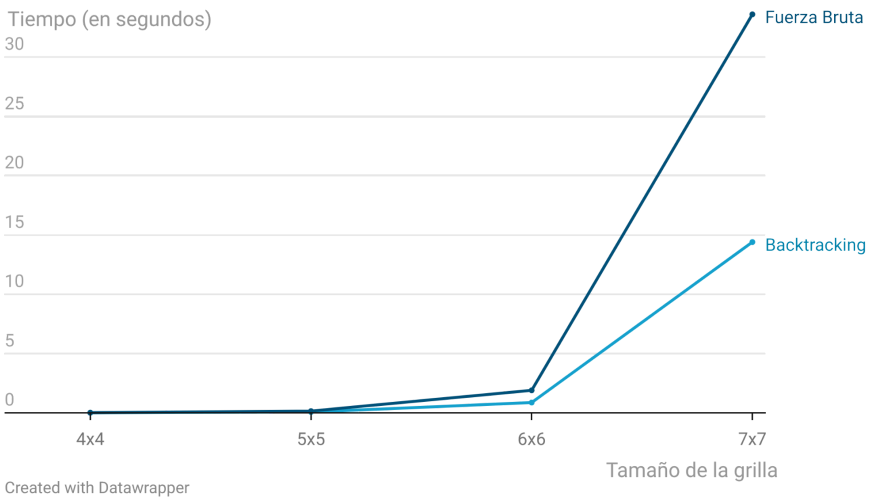
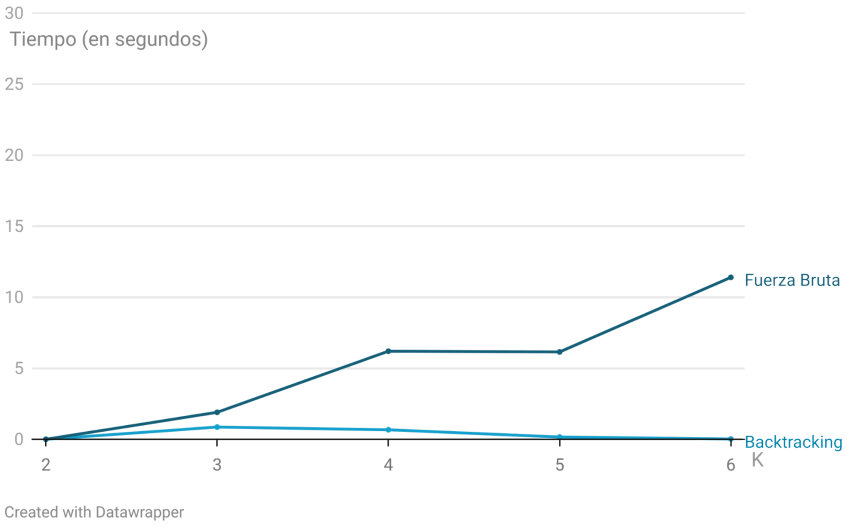


Figura 3.2: Aumento de breakpoints - Performance

Obs: Todos los valores fueron observados para una grilla de 6x6, instancia Titanium.



4. Conclusiones

- Propuesta: En base a los análisis previos, ¿cuál sería la sugerencia del grupo respecto al algoritmo, implementación y lenguaje de programación?

Como conclusión, el proceso de ajuste de parámetros y la selección de la granularidad de la discretización y la cantidad de breakpoints son aspectos críticos para lograr aproximaciones precisas y optimizar la performance de los algoritmos. Sin embargo, no existe una única forma eficiente de aproximar los datos, ya que el error y la performance varían según la naturaleza de los datos, su cantidad, distribución, entre otros factores. No hay una solución universal que se pueda generalizar.

Un hallazgo consistente es que el lenguaje de programación C++ tiene un mejor rendimiento que Python en términos de tiempo de ejecución, independientemente de la instancia analizada. Por lo tanto, para mejorar la performance, es extremadamente útil C++ junto con otros enfoques.

Por último, es esencial llevar a cabo experimentos exhaustivos y analizar detenidamente los resultados para optimizar tanto la precisión de las aproximaciones como el tiempo de ejecución de los algoritmos en diferentes contextos y conjuntos de datos, ya que cada situación puede requerir un enfoque específico y adaptado a sus características particulares.

5. Instrucciones de compilación y ejecución

Python

Dependencias

Desde la terminal descargar la librería “**numpy**”.

```
pip install numpy
```

Estructura del proyecto

- python/
    - auxiliares.py // implementación de la función **error**
    - backtracking.py // implementación de la función **backtracking**
    - dp.py // implementación de la función **programación dinámica**
    - fuerza_bruta.py // implementación de la función **fuerza bruta**
    - menu_desplegable.py // menu interactivo para probar funciones y parámetros
    - main.py // contiene la función **tiempo_promedio**, útil para calcular el tiempo promedio de ejecución para distintos parámetros e instancias.

Ejecución

En la terminal ubicarse en "*tp1-td5",* correr el archivo "menu_desplegable.py" y seguir las instrucciones del menú, seleccionando desde la terminal.

```
PS C:\Users\TDV\tp1-td5> python3 src\python\menu_desplegable.py
```

C++

Estructura del proyecto

- cpp/
    - auxiliares.cpp // implementación de la función **error** y **tiempo_promedio**
    - backtracking.cpp // implementación de la función **backtracking**
    - dp.cpp // implementación de la función **programación dinámica**
    - fuerza_bruta.cpp // implementación de la función **fuerza bruta**
    - menu_desplegable.cpp // menú interactivo para probar funciones y parámetros
    - main.cpp // contiene la función **tiempo_promedio**, útil para calcular el tiempo promedio de ejecución para distintos parámetros e instancias.

Ejecución

En la terminal ubicarse en "*tp1-td5\src\cpp*", ejecutar 'make' y después './pwl_fit'.

Seguir las instrucciones del menú desplegable.

```
PS C:\Users\TDV\tp1-td5> cd src\cpp
PS C:\Users\TDV\tp1-td5\src\cpp> make
...
PS C:\Users\TDV\tp1-td5\src\cpp> ./pwl_fit
...
```