Caracterización y predicción del desempeño en álgebra lineal numérica dispersa mediante aprendizaje automático

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Eduardo González

Tutores: Ernesto Dufrechou
Pablo Ezzatti

2 de agosto de 2021

Resumen

La resolución de sistemas lineales triangulares dispersos (sptrsv) es un componente principal en diversos métodos numéricos, como la resolución de sistemas lineales mediante la factorización LU, o métodos iterativos precondicionados con factorizaciones incompletas. Esto convierte a la sptrsv en una herramienta importante en áreas de la computación como el diseño de simulaciones, modelado de transito, y también en otras disciplinas como matemática, física, economía, entre otras. Es por estas razones que el desarrollo de algoritmos eficientes para este tipo de sistemas es crucial.

Producto de la evolución de las unidades de procesamiento gráfico (o GPUs, por su sigla en inglés), se han desarrollado múltiples algoritmos que permiten aprovechar el paralelismo masivo que este hardware ofrece para la resolución de sistemas lineales dispersos. Si bien las GPUs son especialmente aptas para resolver problemas de álgebra densos, el ancho de banda en el acceso a memoria, típicamente mayor a 10 veces el de la unidad de procesamiento central (o CPU, por su sigla en inglés), y la enorme presencia de GPUs en nodos de cómputo de *datacenters* motiva a explotarlas lo mejor posible en problemas dispersos. Su uso ha proporcionado un gran aumento en la velocidad de resolución de este tipo de problemas en comparación con algoritmos implementados en la CPU, especialmente en los problemas de mayor escala.

Sin embargo, la elección de cuál algoritmo utilizar en cada situación es un problema no trivial, que depende de factores como el formato de representación de la matriz, su tamaño, la proporción de elementos distintos de cero y la ubicación de los mismos. Se busca entonces diseñar un mecanismo automático para seleccionar el *solver* adecuado para cada matriz. Esto es útil sobre todo en el contexto de métodos iterativos, donde se resuelvan decenas o cientos de sistemas para cada patrón de no-ceros (es decir, la familia de matrices cuyas entradas distintas a cero tienen las mismas posiciones, si bien el valor de las entradas puede variar). Además, esta elección debe realizarse de forma computacionalmente rápida, ya que de otra forma la ganancia de tiempo obtenida por usar el algoritmo de resolución correcto se vería contrarrestada por el costo del algoritmo de elección. Esta restricción implica que el algoritmo de elección debe ser simple, o al menos no costoso computacionalmente, y que la obtención de datos de la matriz debe ser rápida, debiendo evitarse la lectura de la matriz entera.

En este proyecto se proponen diversas estrategias para mejorar los procesos de selección de métodos de resolución (*solvers*) existentes, en particular concentrándose en casos donde un mismo patrón de matrices es utilizado en múltiples sistemas a resolver. Un primer enfoque consiste en estimar la cantidad de niveles (donde un nivel es un conjunto de filas independientes) de la matriz del sistema, para luego utilizar este valor en algoritmos de aprendizaje automático para predecir el *solver* óptimo, concentrándose en la precisión y velocidad del algoritmo estimador. Otra temática abordada es el impacto real de los niveles en la predicción de solver óptimo, para esto se utilizan matrices generadas automáticamente para entrenar algoritmos de aprendizaje automático.

Con los resultados de estos estudios se pudo concluir que la cantidad de niveles influye al *solver* óptimo de forma menor pero significativa, y se desarrollaron heurísticas de estimación de niveles con buena precisión y velocidad, gracias al uso de tarjetas gráficas.

Tabla de contenidos

1. Introducción

Resolver sistemas lineales triangulares como parte de una aplicación en el mundo real requiere eficiencia computacional, ya que esta operación suele realizarse dentro de un bucle dentro del programa, por ejemplo como componente de métodos iterativos, lo cual hace que una resolución ineficiente enlentezca todo el programa.

Un ejemplo de aplicación de sistemas lineales son las tomografías computarizadas de rayos X [15]. La tomografía computarizada es una técnica que surge del problema de pérdida de información de profundidad en rayos X tradicionales, en donde se proyecta la estructura del cuerpo en dos dimensiones. Las tomografías computarizadas resuelven este problema lanzando rayos desde varias direcciones, y luego reconstruyendo imágenes seccionales del cuerpo utilizando la información obtenida. Esta reconstrucción requiere un uso intensivo de sistemas dispersos, lo cual puede suponer un costo computacional elevado. Otro ejemplo se da en el campo de dinámicas de fluidos. Librerías dedicadas a este propósito tienen sistemas lineales dispersos como una de sus operaciones más costosas desde el punto de vista computacional, especialmente en geometrías complejas. Una de las librerías dedicadas a dinámicas de fluidos es OpenFOAM [34], la cual a su vez es utilizada en múltiples aplicaciones. En particular, ha sido utilizada en un estudio [18] para modelar la generación de energía eólica en entornos urbanos enfocado en la situación energética de Uruguay.

En algunas de las aplicaciones de resolución de sistemas lineales, incluidas las dos mencionadas anteriormente, se sabe de antemano que la matriz de entrada es siempre dispersa (esto es, que una gran proporción de sus entradas son cero), lo cual requiere que se utilicen algoritmos que eviten realizar operaciones sobre las entradas de la matriz que sean cero, pues no afectan la solución del sistema.

Se trabaja desde hace mucho tiempo en resolver sistemas lineales triangulares de este tipo de forma paralela, es decir, aplicando varios procesadores a la resolución del mismo. Entre estos algoritmos se destacan dos paradigmas distintos: los que utilizan una etapa de pre-procesamiento y los que no. Los algoritmos en el primer paradigma suelen calcular los "niveles" de la matriz, que son particiones de filas de la matriz que pueden ser procesadas en paralelo, mientras que los algoritmos pertenecientes al segundo paradigma procesan la matriz sin necesidad de conocer estas particiones.

Existen múltiples algoritmos en ambas categorías, algunos de los cuales son discutidos en este informe, y mediante previos estudios (por ejemplo [14]) acerca del desempeño de los mismos se ha podido determinar que ningún algoritmo supera a los demás en todos los escenarios. Esto motiva al desarrollo de un mecanismo que permita predecir cuál rutina es más eficiente en cada sistema particular.

Además, estudios previos con aprendizaje automático indican que uno de los factores más importantes que determinan cuál algoritmo es el más rápido en un sistema determinado es la cantidad de niveles de su matriz correspondiente, que como ya se mencionó está relacionada fuertemente con el potencial de paralelización del sistema. Esto es importante, ya que muchos de los algoritmos que resuelven sistemas lineales dispersos triangulares son capaces de sacar partido de la ejecución en tarjetas gráficas, o GPUs, las cuales son diseñadas para ejecutar código de forma masivamente paralela,

sacrificando para este propósito velocidad al ejecutar código secuencial, al compararlas con las unidades de procesamiento centrales (CPUs). Obtener la cantidad exacta de niveles de una matriz es muy costoso para ser utilizado de forma práctica para decidir el algoritmo a utilizar (por ejemplo con aprendizaje automático), ya que obtener este valor es equivalente en tiempo a resolver el sistema mediante un enfoque con etapa de pre-procesamiento.

Uno de los problemas que se plantea atacar en este proyecto es la selección automática del algoritmo de resolución de sistemas para una matriz determinada. Para esto se plantea la utilización de algoritmos de aprendizaje automático, y en particular la utilización de la suite de aprendizaje automático Classification Learner App [42], provista por MATLAB, para decidir cuáles características de la matriz son importantes para determinar el mejor algoritmo, y a partir de esto generar un algoritmo clasificador con un nivel de precisión alto. Para este propósito se propone también el uso de matrices generadas de forma automática, para lograr obtener una cantidad suficiente de matrices con las que entrenar estos algoritmos.

Otro de los problemas que se plantea resolver es la obtención de la cantidad de niveles de la matriz. Una heurística que permite aproximar la cantidad de niveles fue desarrollada en 2019 [28], pero el error de la aproximación puede llegar a ser demasiado alto en algunas matrices como para ser útil. Además, al estar implementada en CPU esta heurística sufre de no poder tomar ventaja de la gran capacidad de paralelismo ofrecidas por otras arquitecturas de hardware, como las GPUs. En este informe se proponen diversas variantes de la heurística original, ofreciendo opciones en GPU que incrementan la velocidad de la misma, y algunas modificaciones para intentar obtener aproximaciones más precisas.

El resto del informe se organiza de la siguiente manera. En la Sección 2 se ofrecen conceptos preliminares que brindan un marco teórico a la propuesta. En particular se discuten las características de las unidades de procesamiento gráfico que son utilizadas en este trabajo, las matrices dispersas y sus varios formatos de almacenamiento, los diferentes métodos de resolución de sistemas dispersos, y algunos conceptos básicos de aprendizaje automático.

En la Sección 3 se presentan propuestas para estimar la cantidad de niveles de cada matriz de forma rápida, lo cual puede ser utilizado como una entrada a algoritmos de aprendizaje automático, como los que fueron utilizados en este proyecto. Luego, estas nuevas heurísticas son puestas a prueba en un conjunto de matrices de distintas características, basadas en problemas reales, y se analizan tanto la precisión alcanzada como su tiempo de ejecución.

A continuación, en la Sección 4 se analiza el impacto de los niveles a la hora de seleccionar el algoritmo de resolución más eficiente para cada sistema, y se discute una propuesta de utilizar aprendizaje automático para seleccionar el mejor algoritmo de resolución de sistemas lineales dispersos según las características de cada sistema, basándose en un estudio previo, y utilizando matrices generadas automáticamente para alcanzar conjuntos de datos de una dimensión elevada, y de esta forma obtener resultados más precisos. Por último, en la Sección 5 se discuten los resultados obtenidos, así como posibles temas relacionados a abordar en futuras investigaciones.

2. Conceptos preliminares

En esta sección se discuten conceptos preliminares, relacionados a las propuestas presentadas, ya sea por ser utilizados por dichas propuestas, o por ser procesos que las propuestas buscan mejorar.

Los conceptos presentados en esta sección son las matrices dispersas, qué son y cómo son almacenadas, la resolución de sistemas lineales triangulares dispersos, analizando los algoritmos más comunes, las GPUs, analizando su historia, funcionamiento y el uso de las mismas para resolver sistemas lineales triangulares dispersos, analizando los algoritmos más comunes. Finalmente se ofrecen conceptos básicos sobre aprendizaje automático, su historia, y un análisis de los algoritmos más comunes en este campo.

2.1. Matrices dispersas

En forma intuitiva, una matriz dispersa o rala (en inglés *sparse*) es aquella que tiene una gran proporción de sus coeficientes iguales a cero o a cierto valor específico (en este caso las matrices pueden ser convertidas a una matriz dispersa tradicional restándole dicho valor a todas las entradas de la matriz). Este tipo de matrices puede ser usada para representar la adyacencia de nodos en grafos donde cada uno está relacionado con una cantidad reducida de nodos, como por ejemplo en redes sociales o intersecciones de calles.

En general, los algoritmos de matrices densas son muy ineficientes al ser aplicados a matrices dispersas. Por un lado, debido al incremento en el uso de memoria, y por otro lado debido a la gran cantidad de operaciones redundantes (involucrando las entradas con valor cero de la matriz). Estas razones han motivado el desarrollo de diversos formatos de almacenamiento y varios algoritmos específicos para tratar con matrices almacenadas en dichos formatos.

Los formatos de almacenamiento disperso buscan almacenar una mínima cantidad de entradas con valor igual a cero (idealmente ninguna), ya que estas pueden ser inferidas a partir de las entradas que no aparecen en la matriz almacenada. Específicamente se emplean distintas técnicas para representar las posiciones de las entradas con elementos distintos de cero dentro de la matriz. A continuación se resumen algunos de los formatos más comúnmente utilizados.

2.1.1. Formato COO

El formato COO (del inglés *COOrdinate list*)[4] consiste en almacenar la matriz con 3 vectores, un vector *data* que almacena los valores distintos de cero, ordenados por su posición en la matriz original, por fila primero y por columna luego. Otros dos vectores del mismo largo, *row* y *column* son utilizados para almacenar la fila y la columna respectivamente de cada uno de estos valores en la matriz.

Por ejemplo, la matriz:

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 3 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

sería representada con los siguientes vectores:

$$data = \begin{bmatrix} 4 & 5 & 3 & 2 & 1 & 1 \end{bmatrix},$$
$$row = \begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 3 \end{bmatrix},$$
$$column = \begin{bmatrix} 0 & 1 & 0 & 2 & 1 & 3 \end{bmatrix}.$$

Este formato tiene como ventaja principal la facilidad de modificación de la matriz, manteniendo una velocidad de acceso aleatorio razonable si se ordenan las entradas por fila y columna.

2.1.2. Formato CSR

El formato CSR (del inglés *Compressed Sparse Row*)[6] consiste en almacenar la matriz con 3 vectores, un vector *data* para los valores distintos de cero, ordenados por su posición en la matriz original, por fila primero y por columna luego, de forma idéntica al formato COO. Un segundo vector del mismo largo *indices_col* es utilizado para almacenar la columna de cada uno de estos valores en la matriz. Para conocer la fila de cada valor se utiliza un tercer vector ptr_fil, de largo igual a la cantidad de filas más uno, el cual contiene el número acumulado de valores distintos de cero al comienzo de cada fila, y el número total de valores distintos de cero de la matriz (o el largo de los dos vectores anteriores) como valor final. Este formato tiene la ventaja de requerir menos espacio para almacenar las posiciones de las filas, bajo el supuesto de matrices con más de un elemento por fila en promedio, ya que el largo del tercer vector es igual al número de filas, y no al de valores distintos de cero como en el caso del formato COO.

Por ejemplo, la matriz:

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 3 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

sería representada con los siguientes vectores:

$$data = \begin{bmatrix} 4 & 5 & 3 & 2 & 1 & 1 \end{bmatrix},$$
$$indices\_col = \begin{bmatrix} 0 & 1 & 0 & 2 & 1 & 3 \end{bmatrix},$$
$$ptr\_fil = \begin{bmatrix} 0 & 1 & 2 & 4 & 6 \end{bmatrix}.$$

Este formato es posiblemente el más utilizado, y se especializa en acceso rápido a las filas de una matriz, lo cual es especialmente útil en la operación de multiplicación matriz-vector.

2.1.3. Formato CSC

El formato CSC (del inglés *Compressed Sparse Column*)[6] es muy similar al anterior, y de hecho consiste en la representación CSR de la transpuesta de la matriz original.

Por ejemplo, la matriz:

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 3 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

sería representada con los siguientes vectores:

$$\begin{aligned} data &= \begin{bmatrix} 4 & 3 & 5 & 1 & 2 & 1 \end{bmatrix}, \\ indices\_fil &= \begin{bmatrix} 0 & 2 & 1 & 3 & 2 & 3 \end{bmatrix}, \\ ptr\_col &= \begin{bmatrix} 0 & 2 & 4 & 5 & 6 \end{bmatrix}. \end{aligned}$$

Este formato, de forma análoga a CSR, se especializa en acceso rápido a las columnas de la matriz, y por lo tanto es muy útil en multiplicaciones vector-matriz.

2.1.4. Formato Ellpack

El formato Ellpack [4] consiste en, utilizando el número máximo de valores distintos de cero de una fila ($nnzr$), crear dos matrices con la misma cantidad de filas que la matriz inicial, y $nnzr$ columnas, donde la primer matriz $data$ contiene los valores distintos de cero de cada fila, llenando los lugares sobrantes con 0, y la segunda matriz $indices$ guarda la posición de la columna de cada valor. Este formato es, potencialmente, más ineficiente que CSR en cuanto al volumen de almacenamiento, debido a que podría llegar a guardar muchos ceros si la cantidad de valores distintos a cero en cada fila es muy variable. Sin embargo, en general es más eficiente que otros formatos en arquitecturas vectoriales (SIMD), como las basadas en GPUs, debido a la disposición de los valores. Por esta razón, matrices en este formato suelen ser almacenadas por columnas para permitir el procesamiento de varias filas en paralelo, permitiendo que cada vector de hilos acceda a posiciones contiguas en cada paso.

Por ejemplo, la matriz:

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 3 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

sería representada con las siguientes matrices:

$$data = \begin{bmatrix} 4 & 0 \\ 5 & 0 \\ 3 & 2 \\ 1 & 1 \end{bmatrix},$$

$$indices = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 2 \\ 1 & 3 \end{bmatrix}.$$

2.1.5. Formato DIA

El formato DIA [4] (DIAgonal) está diseñado para almacenar matrices donde una gran proporción de los coeficientes distintos de cero están alojados en unas pocas diagonales, y consiste en guardar cada diagonal con al menos un valor distinto de cero como una fila en una matriz *data* (con los valores faltantes de las diagonales no centrales llenados con 0). Para recuperar el índice de cada diagonal se utiliza un vector *offsets* de largo igual a la cantidad de diagonales con valores distintos de cero, donde el valor 0 implica que la diagonal correspondiente es la central, un valor positivo d implica que está d diagonales arriba de la central, y uno negativo que está $-d$ abajo de la diagonal central.

Por ejemplo, la matriz:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 3 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 1 \end{bmatrix},$$

sería representada con la siguiente matriz:

$$data = \begin{bmatrix} 0 & 3 & 3 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 2 \end{bmatrix}.$$

Y el vector de índices:

$$offsets = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}.$$

2.1.6. Otros formatos de almacenamiento

Existen otros formatos menos comunes para almacenar matrices. Estos suelen ser variantes de los formatos presentados anteriormente y se utilizan en situaciones más específicas donde el tipo de matrices utilizadas o los algoritmos aplicados a éstas son favorecidos por su utilización. Entre ellos se destacan:

1. HYB El formato HYB (*HYBrid*) es, como su nombre lo indica, un híbrido de otros dos formatos, típicamente Ellpack y COO. Este formato busca resolver el problema de tener una gran variación en el largo de las filas, lo cual hace que el formato Ellpack puro almacene una gran cantidad de ceros. Esto se logra almacenando K columnas en formato Ellpack, y el resto en formato COO, con un cálculo de costos computacionales para determinar K [4].

2. BCSR (*Block CSR*) es un tipo de formato de almacenamiento de matriz por bloque, con bloques de tamaño fijo, donde el conjunto de bloques que contienen uno o más

elementos distintos de cero se almacena utilizando el formato CSR, y el contenido de cada bloque se almacena utilizando un formato denso [13].

3. SELL-P (*Sliced Padded ELLPACK*) es un formato derivado del formato Ellpack, cuya diferencia radica en dividir las filas de la matriz en segmentos, de forma que las filas pertenecientes a cada segmento contengan aproximadamente la misma cantidad de elementos distintos de cero. De esta forma es posible utilizar menos ceros para realizar el *padding* de cada *slice*. El largo de cada segmento es un múltiplo de la cantidad de hilos utilizados en cada fila [3].

4. JDS El formato JDS (*Jagged Diagonal Storage*) deriva del formato CSC, y consiste en reordenar las filas según el número de entradas distintas de cero en las mismas (guardando un vector de permutaciones), y almacenando los valores y posiciones en las filas de los mismos (como en CSC). Por último, los valores son movidos a la izquierda de la matriz y se computa el vector de punteros de CSC [7].

2.1.7. Formatos de almacenamiento en disco

Las matrices dispersas utilizan formatos de almacenamiento que se adecúan a su naturaleza dispersa para reducir el tamaño de las mismas en disco. Los formatos más utilizados son los siguientes:

1. MatrixMarket: Este formato [29] se basa en el formato COO de almacenamiento de matrices dispersas. Las entradas distintas de cero de la matriz se almacenan en texto plano, junto a sus coordenadas, una entrada por línea, y un encabezado al inicio del archivo determina el tipo de dato almacenado en la matriz (punto flotante, entero o booleano), así como su estructura de simetría (lo cual permite ahorrar espacio en el archivo).

2. Matlab: Este formato se basa en el formato de almacenamiento COO, al igual que el formato MatrixMarket pero, a diferencia de éste, es un formato de archivo binario, el cual almacena la matriz tal y como se almacena en el programa Matlab. Este archivo es comprimido utilizando un algoritmo de compresión.

3. Rutheford-Boeing: El formato Rutheford-Boeing es un formato derivado del formato Harwell Boeing [22], el cual es un formato de almacenamiento en texto plano que almacena las matrices en formato CSC. Al igual que el formato MatrixMarket, tiene opciones para decidir el tipo y estructura de simetría de la matriz, así como para almacenar un término independiente para cierto sistema de ecuaciones.

2.2. Resolución de sistemas lineales triangulares dispersos

Un *solver* de sistemas lineales triangulares dispersos es un algoritmo computacional que permite resolver sistemas de este tipo, generalmente de forma exacta (a menos de errores de representación en punto flotante). Si bien métodos iterativos también han sido propuestos [2], éstos no han tenido gran desarrollo en el caso general. Es importante

notar que, si bien los *solvers* son independientes del formato de almacenamiento utilizado, su performance puede verse fuertemente afectada según si se utiliza un formato de almacenamiento u otro.

2.2.1. Algoritmo de sustitución

El algoritmo de sustitución, o *forward-substitution*, es el método usual para resolver sistemas lineales triangulares de la forma:

$$Lx = b,$$

donde $L \in \mathbb{R}^{n \times n}$ es una matriz triangular inferior (dispersa), $b \in \mathbb{R}^n$ es el término independiente del sistema (RHS por su sigla en inglés) y $x \in \mathbb{R}^n$ es el vector de incógnitas. Este algoritmo consiste en multiplicar los coeficientes distintos de cero de cada fila de L por su correspondiente valor (ya encontrado) de x. Hecho esto, se suma el resultado al valor correspondiente de b y se divide esto último por el valor correspondiente de la diagonal de L, lo cual resulta en el nuevo valor de x para esa fila. En el Algoritmo 1 se describe el procedimiento para una matriz almacenada en formato CSR.

Algoritmo 1 Solución secuencial de sistemas triangulares inferiores para matrices dispersas almacenadas en formato CSR utilizando *forward-substitution*

Input

$data$	Arreglo de entradas de la matriz en formato CSR
$indices\_col$	Arreglo de índices de columnas de la matriz en formato CSR
ptr_fil	Arreglo de punteros a filas de la matriz en formato CSR
b	Término independiente del sistema a resolver

Output

x	Vector solución de la matriz

$b = x$
for $i = 0$ **to** $len(ptr\_fil) - 1$ **do**
  for $j = ptr\_fil[i]$ **to** $ptr\_fil[i + 1] - 2$ **do**
    $x[i] = x[i] - data[j] * x[indices\_col[j]]$
  end for
  $x[i] = x[i]/data[ptr\_fil[i + 1] - 1]$
end for
return x

2.2.2. Resolución por niveles

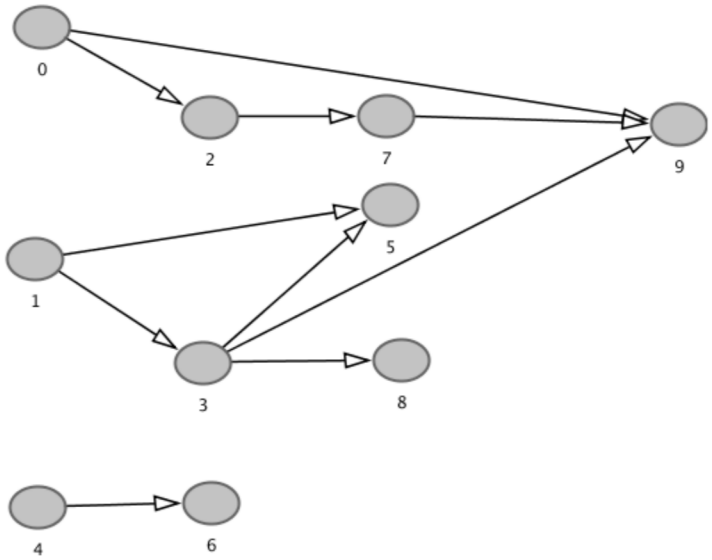
En el método antes presentado, cada entrada del vector x debe ser computada luego de que todas las incógnitas de las que su correspondiente fila depende hayan sido resueltas. Por lo tanto, en el caso disperso se puede construir un grafo dirigido acíclico (DAG, por su sigla en inglés) que represente las dependencias entre las incógnitas. Este grafo se construye a partir de las posiciones de los coeficientes distintos de cero de la matriz L.

En la Figura 1 se muestra una matriz dispersa, así como su patrón de no ceros, la cual puede representarse con el grafo dirigido acíclico de la Figura 2.

Figura 1: Matriz dispersa (izquierda) y su correspondiente patrón de no ceros (derecha), con una X representando la posición de un coeficiente distinto de 0.

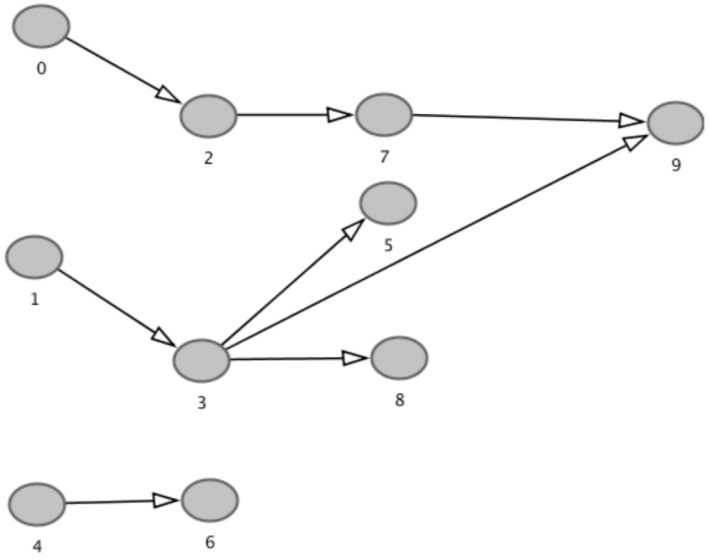
$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 & 0 & 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} X & & & & & & & & & \\ & X & & & & & & & & \\ X & & X & & & & & & & \\ & X & & X & & & & & & \\ & & & & X & & & & & \\ & X & & X & & X & & & & \\ & & & & X & & X & & & \\ & & X & & & & & X & & \\ & & & X & & & & & X & \\ X & & & X & & & & X & & X \end{bmatrix}$$

Figura 2: Grafo de dependencias de niveles de la matriz dispersa de la Figura 1.



El grafo de la Figura 2 puede a su vez ser simplificado. Esto se logra removiendo las dependencias redundantes (por la propiedad transitiva de la relación de dependencia), como puede verse en el grafo de la Figura 3.

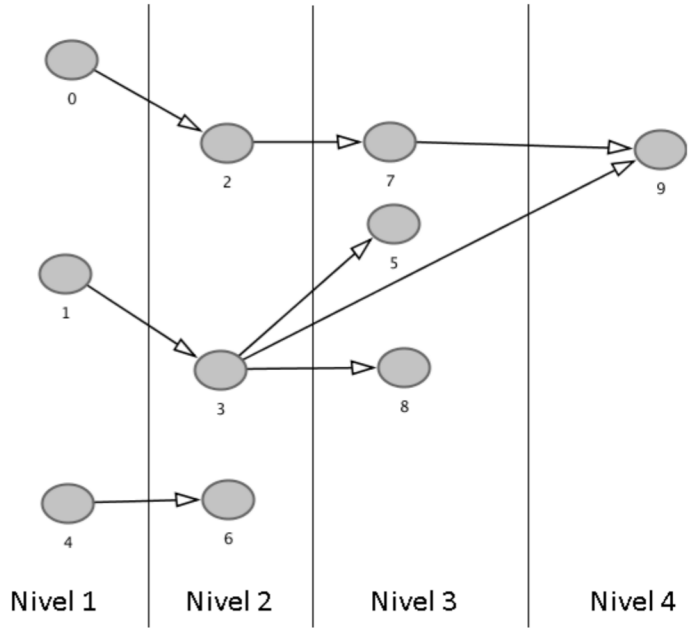
Figura 3: Grafo de dependencias de niveles de la Figura 2, luego de eliminar las dependencias redundantes.



El algoritmo de niveles, o *level-set scheduling*, consiste en asignarle a cada incógnita un número natural, que se denomina nivel, el cual vale 1 si la incógnita no tiene dependencias (o equivalentemente, si es un nodo fuente en el DAG), valiendo 1 más la cantidad de aristas necesarias (en el DAG simplificado) para ir del nodo fuente hasta el nodo correspondiente a la incógnita en los demás casos. El nivel representa la mínima cantidad de iteraciones necesarias para resolver una determinada incógnita si en cada iteración se resolviesen todas las incógnitas sin dependencias de forma paralela. De esta forma, una incógnita depende solamente de otras pertenecientes a niveles inferiores, y esto permite calcular todas las incógnitas de cada nivel de forma paralela.

Por ejemplo, partiendo del grafo anterior los niveles serían los representados por la Figura 4.

Figura 4: Grafo de dependencias de niveles de la Figura 3, con sus vértices particionados por niveles.



Este paradigma de resolución de sistemas lineales fue introducido en 1989 para explotar de la mejor manera posible el paralelismo ofrecido por sistemas de varios procesadores con memoria compartida [1]. Específicamente, este algoritmo fue propuesto originalmente para un sistema Alliant FX/8, con 8 procesadores (ver Figura 5).

Figura 5: Fotografía de un sistema Alliant FX/8. La primera propuesta del paradigma de resolución por niveles fue probada en uno de estos sistemas.

2.3. GPUs

Las unidades de procesamiento gráfico (o en inglés *graphics processing units*, o GPUs) son dispositivos de hardware creados originalmente para acelerar la representación de gráficos en sistemas de computación en tiempo real, y en particular para su utilización en videojuegos. En la última década se ha vuelto cada vez más habitual la utilización de las mismas en computación de propósito general, debido a su gran capacidad de cómputo en paralelo. Un claro ejemplo es la amplia adopción de estos dispositivos para el entrenamiento de redes neuronales [40].

2.3.1. Historia

Las GPUs fueron originalmente creadas para asistir en el procesado en tiempo real de gráficos, a partir de los 90s. Antes de eso, la tarea de dibujar gráficos en la pantalla se hacía en la CPU lo cual, debido a su arquitectura, no era eficiente (tanto por su naturaleza secuencial como por el hecho de que era un recurso compartido por la lógica del programa y por la representación en pantalla).

Las primeras GPUs eran muy limitadas en cuanto a su programabilidad, con pocas técnicas posibles, cada una implementada modificando directamente el hardware. Sin embargo, a medida que la velocidad de las GPUs creció se fueron agregando funcionalidades que otorgaban al programador mayor control sobre los efectos gráficos. Esto hizo que las GPU fueran abandonando las técnicas especializadas y se orientaran a una arquitectura más acorde a la programación general. Esto evolucionó en el desarrollo de *shaders* que son pequeños programas que se ejecutan en la GPU y que permiten al programador decidir exactamente como se ve cada vértice o pixel en una escena en tres dimensiones [38].

Debido a la gran capacidad de cómputo de las GPUs, éstas comenzaron a utilizarse en aplicaciones no relacionadas a la computación gráfica. Esto se vio acentuado con la explosión, en la última década, de algoritmos de *deep learning*, es decir, redes neuronales con una gran profundidad que son entrenadas en GPUs [24].

Inicialmente se utilizaron *compute shaders* para la programación de propósito general, los cuales son *shaders* que se ejecutan fuera del *pipeline* de renderizado y que operan sobre información arbitraria. Sin embargo esto involucraba operar sobre abstracciones de programación gráfica, como texturas, lo cual complicaba las tareas. Una de las primeras tareas que fue implementada utilizando *shaders* fue la descomposición LU de una matriz [16].

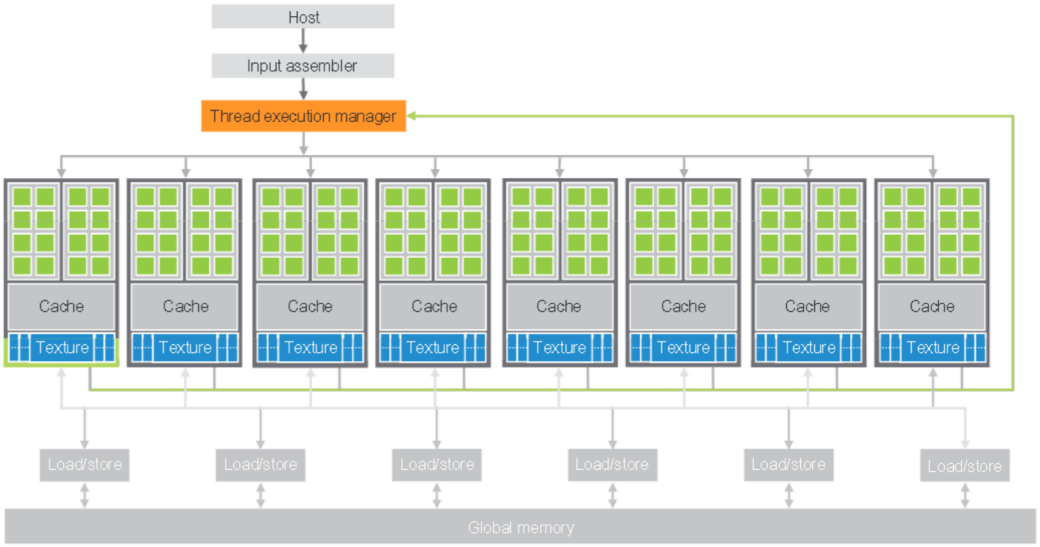
Esto cambió con el desarrollo de CUDA [8] por NVidia. CUDA (*Compute Unified Device Architecture*) es una plataforma de programación de propósito general en GPUs que elimina la necesidad de utilizar *compute shaders*. La popularidad de programación en GPU propició la adopción de estos en centros de computación de alta performance (o HPC por su sigla en inglés), para ser utilizados en tareas con altos requerimientos computacionales, como aprendizaje automático de modelos complejos, pronóstico del tiempo [32], simulaciones de mecánicas de fluidos [20], entre otras aplicaciones.

2.3.2. Arquitectura de Hardware

Una GPU moderna (en particular, una GPU NVidia, las cuales son las más utilizadas por la comunidad científica) está compuesta por un arreglo de multiprocesadores (*streaming multiprocessors* o SMs), como se muestra en la Figura 6. Cada uno de estos multiprocesadores, a su vez, está compuesto de varios procesadores (*CUDA cores*), similares a los núcleos de la CPU pero más simples y en mayor cantidad. Esto hace que, si bien la performance de la CPU es mayor a la de la GPU en tareas secuenciales, la GPU es capaz de alcanzar velocidades mucho mayores en tareas paralelas, ya que tiene una mayor cantidad de transistores dedicados a tareas de cómputo.

Estos multiprocesadores comparten una memoria global, lo cual facilita la realización de tareas en paralelo. Además, existe una memoria denominada "compartida", la cual es compartida por los procesadores dentro de un mismo multiprocesador. Esta memoria es más rápida que la memoria global, pero de menor tamaño y no mantiene información entre ejecuciones de programas.

Figura 6: Arquitectura de una GPU moderna (imagen sacada de *Programming Massively Parallel Processors: A Hands-On Approach* [26], segunda edición, página 9)



2.3.3. Programación de propósito general en GPUs

En CUDA se pueden crear funciones (llamadas *kernels*) en un lenguaje similar a C/C++ que se ejecutan en múltiples hilos de la GPU simultáneamente. Estos *kernels*, en general, son llamados por un programa que es ejecutado en la CPU. Los hilos en un *kernel* son agrupados en bloques, los cuales comparten los recursos de un mismo multiprocesador. Estos bloques no pueden tener más de 1024 hilos. Los hilos dentro de un bloque comparten un espacio de memoria de baja latencia llamado "memoria

compartida" y son capaces de sincronizarse durante la ejecución de un *kernel* mediante el uso de primitivas, sin necesidad de devolverle control a la CPU.

Además, los hilos dentro de un bloque son agrupados en grupos de 32 llamados *warps*. En arquitecturas de GPU previas a la generación Volta (2018), los hilos en un *warp* ejecutan en cada instante de tiempo la misma instrucción, lo cual permite utilizar instrucciones especiales para mover datos entre los procesadores de un *warp* de forma paralela sin necesidad de recurrir a operaciones de sincronización. A los hilos dentro de un *warp* se los conoce como *lanes*.

Existe un número de versión en las tarjetas gráficas, llamado *CUDA compute capability*, el cual refiere a las funcionalidades de CUDA que la tarjeta gráfica soporta. Mientras mayor sea, más funcionalidades ofrece y por lo tanto el compilador es capaz de generar código más optimizado.

2.4. Resolución de sistemas triangulares dispersos en GPUs

A continuación se describen los principales esfuerzos de desarrollo de *solvers* para sistemas lineales dispersos triangulares en GPU.

2.4.1. Variante por niveles en GPU

En 2011 se desarrolló un *solver* basado en el paradigma *level-set* que funciona de forma paralela en tarjetas gráficas (GPUs) [30]. En el mismo se utiliza una variante del algoritmo de Kahn [25] para determinar el nivel al que pertenece cada fila, y se utiliza el paralelismo ofrecido por la GPU para encontrar los nodos fuente y remover las aristas adyacentes de los mismos de forma concurrente. El algoritmo también almacena una lista de los candidatos a nodos fuente luego de resolver cada nivel, de forma de reducir la cantidad de nodos a analizar en el siguiente nivel. El autor discute las ventajas y desventajas de utilizar solamente uno o varios bloques de hilos en cada nivel durante la etapa de resolución del sistema, y concluye que esto depende fuertemente de la cantidad de nodos que pertenezcan a cada nivel. Los niveles que tengan más nodos se benefician de la mayor cantidad de hilos disponibles que resulta de utilizar múltiples bloques, mientras que niveles con menos nodos no obtienen ningún beneficio al utilizar más hilos. En cambio, procesar estos niveles con un solo bloque de hilos permite utilizar las operaciones de sincronización de bloque en lugar de devolverle el control a la CPU y lanzar un *kernel* por nivel para sincronizar los hilos. Para utilizar la mejor de estas dos metodologías en cada nivel se utiliza una estructura de datos, llamada *chain*, la cual almacena secuencias consecutivas de niveles que puedan ser procesados con un bloque. Esta *chain* es formada dentro de la etapa de análisis de niveles del algoritmo.

2.4.2. Paradigma asincrónico (synchronization-free)

El algoritmo de *sync-free* es similar al anterior en que utiliza una adaptación concurrente del algoritmo de *forward-substitution*. Sin embargo en vez de pre-calcular un grafo de dependencias, el algoritmo lo descubre durante la propia resolución del sistema usando las posiciones de los elementos distintos de cero de la matriz como un grafo

de dependencias, y emplea algún método para determinar cuándo una fila finaliza su ejecución (por ejemplo un vector de booleanos inicializados en falso y actualizados a verdadero cuando una incógnita es resuelta). De esa forma puede determinar el flujo de ejecución dinámicamente, explotando la técnica de espera activa (o *busy waiting*) en las filas cuyas dependencias no hayan sido satisfechas.

El primer algoritmo de este tipo fue propuesto en 1986, y fue diseñado para correr en un sistema multiprocesador con memoria compartida (Sequent Balance 8000)[17]. Este sistema tiene 8 procesadores conectados por un bus, los cuales comparten 8 MB de memoria compartida. Para implementar el algoritmo se utilizaron de 1 a 7 procesadores (con el octavo reservado para operaciones del sistema operativo) para procesar las filas de la matriz, y se utilizó un vector de enteros interpretados como booleanos para determinar cuando una fila termina su ejecución. El programa fue implementado en el lenguaje de programación Fortran.

2.4.3. Variante sync-free en GPU

Si se utilizan GPUs para implementar este método, una opción posible es asignarle un warp a cada fila, con cada dependencia de la fila asignada a un hilo del warp (utilizando varias pasadas en caso de existir más de 32 dependencias) y sacar partido de las operaciones de *shuffle* (operaciones de *warp* que intercambian una variable de un hilo a otro) para decidir si todas las dependencias fueron halladas. Esto se muestra en el Algoritmo 2.

El primer algoritmo bajo el paradigma *Sync-Free* implementado para tarjetas gráficas fue propuesto en 2016, y opera sobre matrices dispersas en formato CSC [27]. Se utiliza un vector de entradas no nulas de tamaño n, que determina la cantidad de entradas no nulas a computar en cada fila, lo cual es hallado en una etapa de preprocesamiento. Luego, se utiliza un vector distinto para contar la cantidad de entradas procesadas en cada fila, y utilizando estos dos vectores se puede determinar que una fila ha sido procesada cuando sus valores correspondientes en ambos vectores sean iguales. Los valores de las incógnitas resueltas hasta el momento se almacenan en un vector, en memoria global, de forma que sean accesibles por todos los hilos.

Basado en el algoritmo anterior, un algoritmo que opera sobre matrices en formato CSR fue diseñado en 2018 [10]. En lugar del vector de entradas no nulas, esta versión utiliza un vector de booleanos en memoria global, inicializados en *false*, para determinar si una incógnita fue encontrada. Este vector es actualizado por un hilo de la GPU una vez que éste resuelve la incógnita. Esto es posible, ya que la estructura de las matrices almacenadas en formato CSR proporciona la cantidad de entradas no nulas en cada fila, lo cual no ocurre en matrices almacenadas en formato CSC. La propuesta anterior fue extendida en diferentes aspectos para aprovechar de forma más eficiente los recursos de hardware y la estructura de la matriz.

Existen tres variantes:

1. Valor inicial de NaN

   En lugar de utilizar un vector de booleanos como en los ejemplos anteriores, es

Algoritmo 2 Solución paralela en GPUs de sistemas triangulares inferiores para matrices dispersas almacenadas en formato CSR utilizando el paradigma *sync-free*. El vector $ready$ contiene valores booleanos inicializados en falso para determinar si una fila fue resuelta.

Input

$data$	Arreglo de entradas de la matriz en formato CSR
$indices\_col$	Arreglo de índices de columnas de la matriz en formato CSR
ptr_fil	Arreglo de punteros a filas de la matriz en formato CSR
b	Término independiente del sistema a resolver
$ready$	Vector auxiliar de booleanos, inicializados en falso

Output

x	Vector solución de la matriz

$warp$ = identificador global del $warp$
$lane$ = identificador de $lane$
$comienzo\_de\_fila = ptr\_fil[warp]$
$suma\_por\_izquierda = 0$
while not $ready[indices\_col[fila + lane]]$ **do**
    esperar
end while
$suma\_por\_izquierda = suma\_por\_izquierda + data[comienzo\_de\_fila + lane] * x[indices\_col[comienzo\_de\_fila + lane]]$
Reducir suma_por_izquierda dentro del warp
if $lane == 0$ **then**
    $x[warp] = b[warp] - suma\_por\_izquierda/data[ptr[warp + 1] - 1]$
    $ready[warp] = true$
end if
return x

posible inicializar el vector x con un valor NaN [39], para minimizar la utilización de memoria global [11].

2. Pre-cálculo de niveles y re-ordenamiento de hilos

   Además, es posible utilizar un análisis del tipo de *level-set* para ordenar los hilos de tal forma que las filas con menores dependencias sean ejecutadas primero, minimizando así el tiempo que los hilos se mantienen en *busy waiting* esperando por sus dependencias, y por lo tanto degradando el paralelismo del algoritmo. Almacenar el orden de ejecución para ser usado por los hilos requiere un vector de permutación adicional [11].

3. Procesamiento de múltiples filas con un warp

   Por último, es posible que un warp procese más de una fila, cuando las filas tienen pocos coeficientes distintos de cero, mejorando los tiempos en las primeras filas de la matriz, o en matrices con muchas filas con pocos elementos distintos de cero [11].

   Esto se logra dividiendo cada warp en particiones iguales. Cada partición de un warp procesa una fila distinta. Luego, se ordenan las filas según la cantidad de elementos distintos de cero. Por último, a cada warp se le asignan filas de forma que procese 32 filas con un elemento distinto de cero (fuera de la diagonal), 16 filas con 2 o menos, 8 filas con 4 o menos, 4 filas con 8 o menos, 2 filas con 16 o menos o 1 fila con 32 o menos (un warp contiene 32 hilos).

2.4.4. Selección automática entre las distintas variantes

Dado que existen diversos solvers, donde cada uno alcanza buen desempeño para algunos sistemas lineales y malo para otros, decidir cuál *solver* utilizar en cada caso es un problema interesante. Esta decisión debe ser rápida, ya que el costo de decidir cuál *solver* utilizar debe ser menor a la ganancia de tiempo obtenida por utilizar el algoritmo seleccionado, en lugar de utilizar el algoritmo que en promedio resuelva mejor todos los sistemas. Este costo se vuelve menos relevante en el caso de que la estructura de la matriz (la posición de los elementos distintos de cero) sea la misma para varios sistemas a resolver. Esta situación es muy común, en especial como parte de métodos iterativos para resolver sistemas lineales generales.

En [12] se han desarrollado heurísticas basadas en inteligencia artificial para determinar cuál *solver* es mejor en cada sistema lineal. Estas heurísticas reciben como parámetro características de la matriz, como la cantidad de elementos distintos de cero, el ancho de banda (la máxima distancia a la diagonal central de un elemento distinto de cero), o la cantidad de niveles de una matriz. Si bien algunos de estos parámetros son "fáciles" de obtener, es decir, se obtienen con un costo computacional acotado (por ejemplo, la cantidad de elementos distintos de cero en una matriz en formato CSR se puede hallar leyendo la última entrada del vector ptr_fil), otros como la cantidad de niveles son demasiado costosos de obtener de forma exacta. Por ejemplo, el costo computacional de obtener la cantidad de niveles es comparable al tiempo de resolución del *solver* con

paradigma *level-set*. Es por esto que se deben utilizar métodos que estimen el valor de estos parámetros de forma eficiente.

2.4.5. Heurística para estimar la cantidad de niveles

La profundidad de un nodo en un DAG está definida como la distancia del mismo al nodo fuente más lejano. De esta forma, los niveles de una matriz pueden definirse como los conjuntos de nodos con una misma profundidad. Basándose en estas definiciones y en el hecho de que las matrices a estudiar son triangulares, es posible deducir que la profundidad de un nodo está acotada por la cantidad de nodos con un índice inferior. Por lo tanto, puede esperarse que los nodos con mayor índice tengan en promedio una mayor profundidad que nodos con índices bajos.

Aprovechando esta observación, en [28] se desarrolló una heurística que estima la cantidad de niveles de una matriz, priorizando en su análisis las filas con índices más altos de la matriz. El algoritmo consiste en explorar un camino de nodos en el DAG correspondiente a la matriz, partiendo desde un nodo de origen y avanzando por los nodos incidentes, prefiriendo aquellos que no sean nodos fuentes, y entre éstos al que tenga un mayor índice. Al llegar a un nodo fuente, se cuentan la cantidad de nodos recorridos y esa es la estimación de la profundidad del nodo original. Estos caminos pueden ser vistos como "rayos" o "disparos", de forma análoga a otras estrategias de computación gráfica y resolución de ecuaciones diferenciales respectivamente.

Como el camino más largo no necesariamente incluye al nodo con mayor índice, el procedimiento se ejecuta en los últimos r nodos del grafo que tengan nodos incidentes y se toma el máximo valor de todas las ejecuciones, esperando que el resultado se acerque al número exacto de niveles al incrementar r, o que por lo menos tienda a un número que permita calcular una aproximación adecuada. Si el nodo inicial de un rayo no tiene nodos incidentes, se considera el nodo con el índice resultante de restarle r al índice anterior, y esto se repite hasta encontrar un nodo con vértices incidentes o hasta que no hayan más nodos.

Una desventaja de este algoritmo es que algunos de los nodos iniciales podrían llegar a ser parte de caminos ya analizados, por lo que no aportarían nueva información y serían un desperdicio en término de cómputos.

El algoritmo siempre devuelve un resultado menor o igual a la cantidad total de niveles en el sistema, ya que calcula de forma exacta la cantidad de niveles de un subgrafo del grafo de dependencias original.

Visualmente, el algoritmo forma "escaleras" en la matriz, partiendo de la fila en la que se lanza el algoritmo, en la diagonal, moviéndose a la izquierda hasta encontrar un valor distinto de cero, luego moviéndose hacia arriba hasta llegar a la diagonal, y repitiendo este proceso hasta llegar a una fila sin dependencias, contando la cantidad de veces que se entró en la diagonal (incluyendo al inicio), como se muestra en la Figura 7.

En la Figura 8 puede verse el mismo lanzamiento desde el punto de vista del grafo de dependencias.

Figura 7: Representación visual en la matriz de la Figura 1 del lanzamiento de un rayo en el algoritmo presentado.

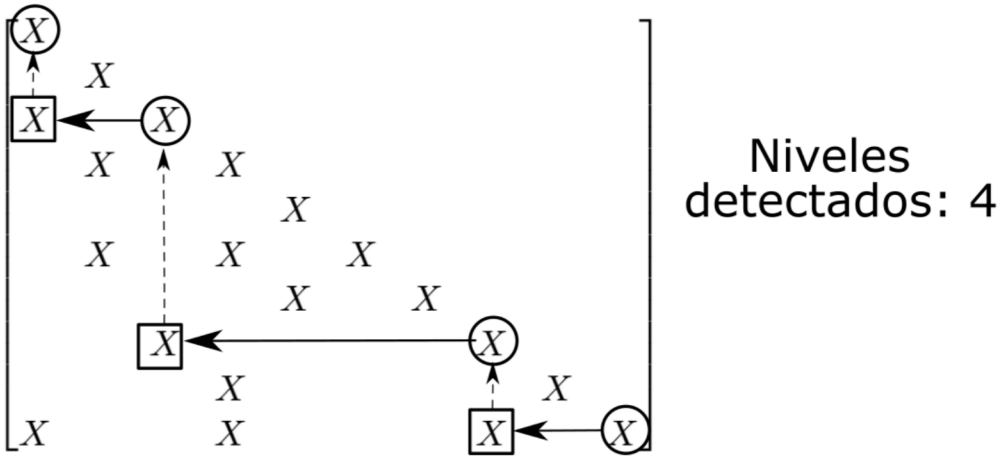
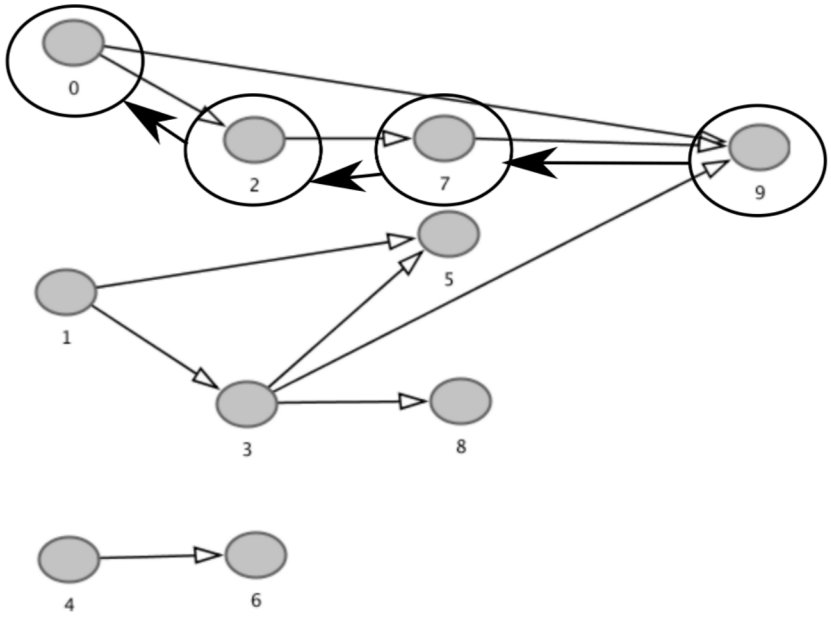


Figura 8: El lanzamiento del rayo de la Figura 7, visto desde el punto de vista del grafo de dependencias (el mismo de la Figura 2).



Una consecuencia de tomar solamente el nodo incidente con mayor índice en cada nodo

es que nunca se toman dependencias que sean redundantes por la propiedad transitiva, ya que siempre se tomaría el nodo intermedio, debido a que tiene un índice más alto. Debido a esto, el lanzamiento mostrado en la Figura 8 no se vería modificado si en lugar de utilizar el grafo de la Figura 2 se utilizara su versión simplificada de la Figura 3.

Algoritmo 3 Algoritmo secuencial de rayos para estimar la cantidad de niveles de una matriz triangular superior dispersa almacenada en formato CSR

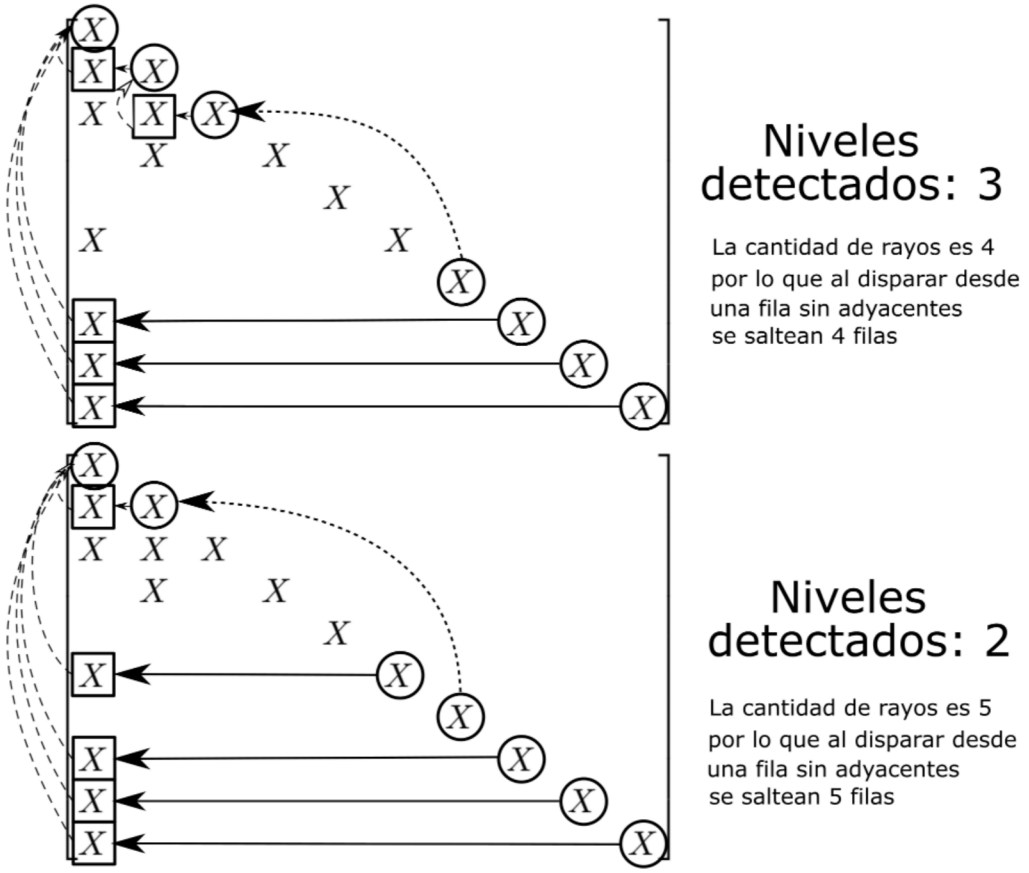
Input
- $data$ — Arreglo de entradas de la matriz en formato CSR
- $indices\_col$ — Arreglo de índices de columnas de la matriz en formato CSR
- ptr_fil — Arreglo de punteros a filas de la matriz en formato CSR
- r — Cantidad de filas iniciales a analizar

Output
- $estimacion\_niveles$ — Estimación de los niveles de la matriz

for $i = 0$ **to** r **do**
  $largo\_camino = 0$
  $fila\_actual = len(ptr\_fil) - i$
  while $fila\_actual$ tiene un solo valor distinto de cero **do**
    $fila\_actual = fila\_actual - r$
  end while
  while $fila\_actual$ tiene más de un valor distinto de cero **do**
    $offset = 2$
    $nnz\_fila =$ cantidad de elementos distintos de cero en $fila\_actual$
    $fila\_siguiente = indices[ptr\_fil[fila\_actual] + nnz\_fila - offset]$
    while $fila\_siguiente$ tiene un solo valor distinto de cero **and** $offset > nnz\_fila + 1$ **do**
      $offset = offset + 1$
      $fila\_siguiente = indices[ptr\_fil[fila\_actual] + nnz\_fila - offset]$
    end while
    $largo\_camino = largo\_camino + 1$
    $fila\_actual = fila\_siguiente$
  end while
  $estimacion\_niveles = max(estimacion\_niveles, largo\_camino)$
end for
return $estimacion\_niveles$

Es interesante notar que la estrategia de saltear filas sin nodos incidentes, salteando r filas cada vez, causa que una ejecución con mayor cantidad de rayos del algoritmo pueda llegar a generar una menor cantidad de niveles que una con menos rayos. Ésto sucede debido a que en el segundo caso el algoritmo podría saltear un nodo sin nodos incidentes y llegar a un nodo que de un nivel alto, mientras que en el primero, este nodo podría no alcanzarse ya que el nodo vacío sería seguido por un nodo distinto, como se demuestra en la Figura 9.

Figura 9: Ejemplo de matriz en la cual un lanzamiento de cinco rayos resulta en una menor cantidad de niveles detectada que un lanzamiento de cuatro rayos.



2.5. Aprendizaje automático

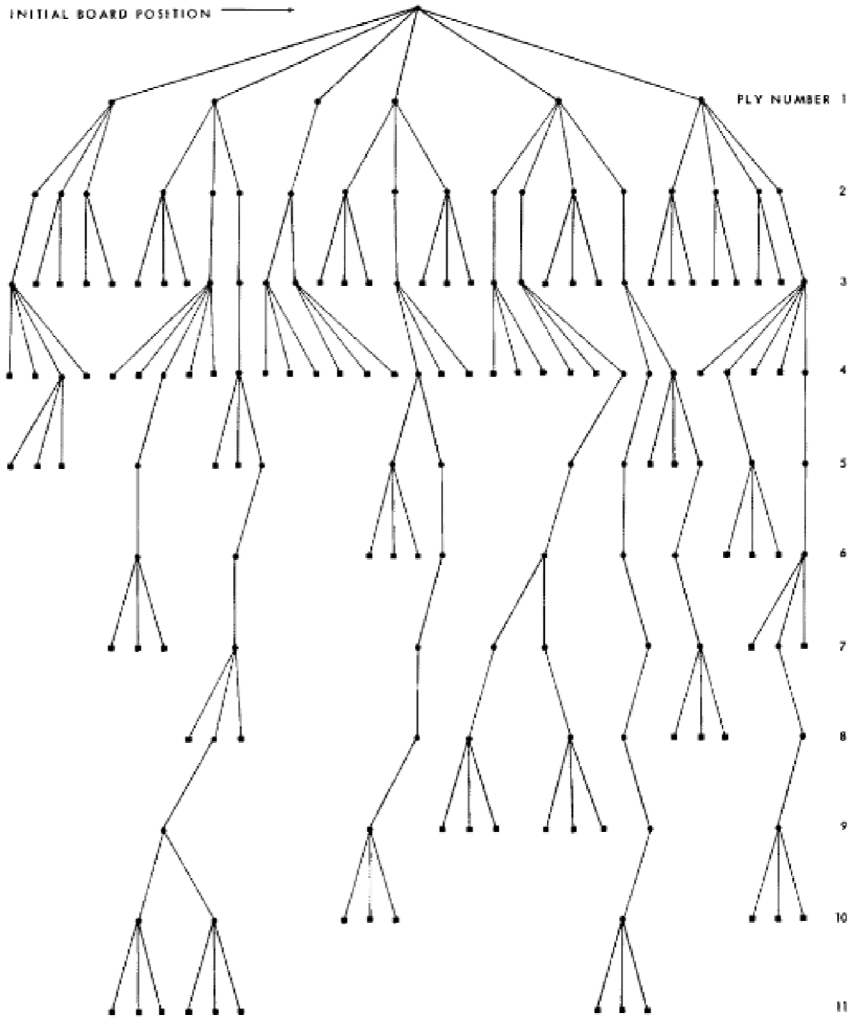
Como se mencionó en la sección anterior, para elegir el mejor *solver* a emplear en un sistema dado se utilizan heurísticas basadas en aprendizaje automático. El aprendizaje automático consiste en algoritmos que "mejoran" a través de la experiencia. Este tipo de técnicas consiste en tomar una serie de elementos e inferir a partir de los mismos características de nuevos elementos, utilizando principalmente conceptos de probabilidad y estadística para lograr este fin.

2.5.1. Historia

El concepto de aprendizaje automático fue acuñado por Arthur Samuel [37] en 1959, aplicado al juego de damas. Este juego fue elegido debido a la simplicidad de sus reglas y al gran tamaño del espacio de solución, lo cual hace imposible un análisis exhaustivo

de las mismas. El algoritmo consistía en, al principio de cada turno, explorar un árbol de posibles movimientos (como se muestra en la Figura 10), tanto propios como del oponente, deteniéndose al acabarse la memoria o al haber pasado un tiempo límite de computo. Luego, las posiciones finales del tablero eran puntuadas y se procedía por el movimiento que haya predicho un mejor resultado.

Figura 10: Representación gráfica del árbol de movimientos en el algoritmo de Arthur Samuel para el juego de damas. Extraído de [37].



En 1958 un artículo [36], por Frank Rosenblatt detallaba un modelo teórico que explica sistemas con "inteligencia" utilizando un sistema nervioso hipotético, utilizando unidades denominadas "perceptrones".

El modelo se basó en las teorías de Thomas Hebb sobre la interacción de neuronas,

publicadas en el libro *The Organization of Behavior* [23] en 1949. Una variante de este modelo es la base del algoritmo de redes neuronales que se usa en la actualidad.

En la última década, la utilización de GPUs para programación de propósito general ha despertado un nuevo interés en el aprendizaje automático, debido a la gran capacidad de paralelismo presente en la mayoría de estos algoritmos, como por ejemplo las redes neuronales.

2.5.2. Tipos de algoritmo

Existen varias formas de categorizar los algoritmos de aprendizaje automático.

Los algoritmos que son de interés en este proyecto pertenecen a la categoría de "aprendizaje supervisado". Estos algoritmos consisten en entrenar un modelo con un conjunto de datos que ha sido previamente anotado con la salida correcta que el modelo debería proporcionar. Luego del entrenamiento, el modelo debería ser capaz de inferir la salida correcta para instancias que no pertenezcan al conjunto de entrenamiento.

Dentro de los algoritmos de aprendizaje supervisado se encuentran al menos dos categorías, en base al objetivo o tarea que realizan: clasificación y regresión.

2.5.2.1. Clasificación

Dado un elemento al cual se le extraen parámetros y un conjunto de grupos, los algoritmos de clasificación son capaces de determinar a cuál de los grupos pertenece el elemento. Por ejemplo, un algoritmo posible podría determinar, dadas varias mediciones médicas, si una persona tiene cáncer o no.

2.5.2.2. Regresión

Otra de las categorías de algoritmos consiste en los algoritmos de regresión. Estos algoritmos consisten en tomar parámetros de un elemento dado, al igual que los algoritmos de clasificación, pero en lugar de asignarles un grupo les otorga valores numéricos, en un dominio potencialmente continuo. Por ejemplo, dadas las mediciones médicas de una persona determinar la probabilidad de desarrollar cáncer en los próximos 5 años.

2.5.3. Algoritmos de aprendizaje automático

A continuación se presentan algunos algoritmos, y se discute su funcionamiento, así como su pertenencia a las categorías discutidas previamente.

2.5.3.1. k-Nearest Neighbors (kNN)

El algoritmo kNN [33] (*k-Nearest Neighbors*, o k vecinos más cercanos en español) es un algoritmo que puede ser utilizado tanto para clasificación como para regresión. Este método consiste en almacenar todas las instancias de aprendizaje utilizadas y, valiéndose de una función de distancia entre dos elementos, evaluar a un nuevo elemento basado en los k elementos más cercanos al mismo. En el caso de clasificación, se clasifica al nuevo elemento de la misma forma que a la mayoría de los k elementos más cercanos,

como puede verse en la Figura 11. En el caso de regresión, se toma el promedio de los k elementos más cercanos. Esto puede apreciarse en la Figura 12.

Figura 11: Ejemplo de clasificación en kNN, con k=3. En este caso, el punto incógnita sería clasificado como un círculo.

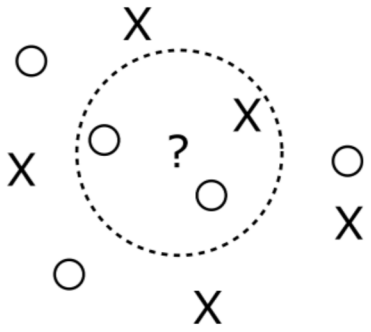
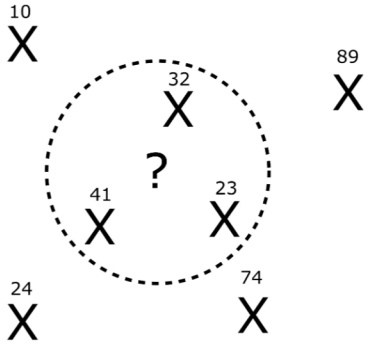


Figura 12: Ejemplo de regresión en kNN, con k=3. En este caso, al punto incógnita le sería asignado el valor 32.



2.5.3.2. Weighted kNN

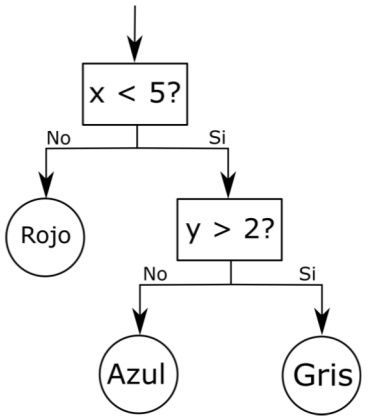
Una variante del algoritmo de kNN clásico [9], este algoritmo consiste en considerar la distancia de los k elementos más cercanos asignándoles un peso mediante la función de distancia del algoritmo kNN. Luego se utiliza este peso para realizar la clasificación o regresión, de forma que las instancias con más peso afecten al resultado de una mayor manera.

2.5.3.3. Árboles de decisión

Los árboles de decisión [31] (*decision trees* en inglés) consisten en árboles, normalmente binarios, que en los nodos hoja contienen una clasificación, y en las ramas contienen un predicado, el cual, dado un elemento, decide a cuál nodo hijo debe ir dicho elemento. De esta forma, al pasarle un elemento al nodo raíz, este llegará eventualmente a un nodo

hoja, el cual contiene la clasificación final del elemento. La construcción de estos árboles de decisión constituye un algoritmo de clasificación de aprendizaje automático. El mismo consiste en elegir un atributo que divida al conjunto de instancias de entrenamiento de forma que los subgrupos sean lo más uniformes posible en términos de su clasificación, crear un nodo con ese predicado, para luego repetir el proceso de forma recursiva hasta llegar a un conjunto de instancias con la misma clasificación, en cuyo caso se construye un nodo hoja. Medidas adicionales pueden emplearse para limitar el tamaño del árbol. Una ventaja de los árboles de decisión es que luego de entrenar un modelo es posible recuperar la información aprendida, ya que el modelo es fácil de comprender por una persona. Un ejemplo de este algoritmo puede verse en la Figura 13.

Figura 13: Ejemplo de árbol de decisión binario. Si se clasificara el punto (x=2, y=3) utilizando este árbol, la clasificación resultante sería "Gris".



2.5.3.4. Empaquetado

Empaquetado [5] (también conocido como *Bootstrap Aggregation*) en inglés, o abreviado a *Bagging* es una técnica que se le aplica a un algoritmo de aprendizaje en un intento de mejorar las predicciones generadas por el mismo. Consiste en, dado un set de datos inicial D de tamaño n, generar m nuevos sets de datos D_i, de tamaño n', tomando n' elementos al azar del set original D, permitiendo repetición. Luego, el algoritmo de aprendizaje se aplica a todos los sets S_i, y los resultados se combinan para obtener la predicción final, ya sea promediando en el caso de regresiones o votando en el caso de clasificaciones.

2.5.4. Overfitting

Los modelos de aprendizaje supervisado se entrenan en instancias conocidas de un problema, para luego ser utilizadas en instancias desconocidas del mismo problema. Sin embargo, puede ocurrir que un modelo presente una precisión mucho menor en instancias desconocidas que en las instancias de entrenamiento. A este fenómeno se lo conoce como "*overfitting*".

Overfitting, o sobreajuste en español, ocurre porque el modelo se ajusta demasiado a las instancias de entrenamiento, ganando precisión en las mismas pero perdiendo generalidad, lo cual empeora su precisión en instancias desconocidas. Este problema ocurre en todos los algoritmos de aprendizaje supervisado, pero los algoritmos más complejos son más susceptibles a este efecto, ya que su mayor nivel de flexibilidad les permite alcanzar mayores niveles de especificidad en sus modelos que otros algoritmos más simples.

Una solución posible es preferir algoritmos más simples, o en el caso de utilizar algoritmos complejos intentar simplificar el modelo resultante.

Además, se debe tener en cuenta el *overfitting* del modelo a la hora de determinar la precisión del mismo. Una solución simple es dejar algunas instancias conocidas fuera del conjunto de entrenamiento, y utilizar estas para evaluar la precisión del modelo entrenado. A este conjunto de instancias se le llama conjunto de *test*.

2.5.4.1. k-fold cross-validation

k-fold cross-validation [35], o KCV, es una de las maneras más populares para estimar la precisión de un modelo de aprendizaje automático. Este método consiste en particionar aleatoriamente las instancias de entrenamiento en k subconjuntos de igual tamaño, y realizar k iteraciones de entrenamiento en las cuales se separa el subconjunto k para evaluar la precisión del modelo, el cual es entrenado con las instancias en los otros subconjuntos. Luego de esto, las k estimaciones obtenidas se combinan de alguna forma, comúnmente promediandolas, para obtener la estimación final.

Este procedimiento ha demostrado resultados menos optimistas (es decir, estimaciones más resistentes a *overfitting*), que separar un subconjunto de test.

3. Mejora de la heurística para estimar la cantidad de niveles

En trabajos anteriores se avanzó en métodos para predecir el desempeño de diversos algoritmos de resolución de sistemas lineales dispersos. En particular, se destacó la importancia de conocer la cantidad de niveles de una matriz. Sin embargo, este valor es computacionalmente costoso de obtener, siendo similar al costo de resolver el sistema directamente, lo cual hace inviable utilizarlo para seleccionar *solvers*. Debido a esto, se desarrolló una heurística para estimar dicha cantidad de forma rápida.

En esta sección se exploran mejoras posibles de esta heurística de estimación de niveles (presentada anteriormente en la Sección 2.4.5), y se estudian las cualidades de las mismas en una colección de matrices dispersas basadas en problemas reales.

3.1. Capacidad de exploración de la heurística original

Uno de los problemas más importantes de la heurística original es la baja precisión en la estimación de niveles. Esto se debe en gran medida a la redundancia en la exploración que puede darse al aumentar la cantidad de lanzamientos del algoritmo, ya que en una gran cantidad de casos los nodos incidentes a un nodo son también los nodos iniciales de otros lanzamientos. En otras palabras, los caminos generados por muchos de los rayos se encuentran contenidos en aquellos generados previamente, lo cual reduce la capacidad de exploración de la heurística, por lo que a continuación se proponen diversas variantes para solucionar este problema e intentar mejorar la precisión del algoritmo de rayos.

3.1.1. Selección aleatoria de nodo incidente (LeSeEs$_{Sr}$)

Esta variante consiste en seleccionar, en lugar del nodo incidente con mayor índice, el nodo incidente r-mayor, con r un número aleatorio entre 1 y R (una constante seleccionada al inicio del algoritmo). Esto permite una mayor dispersión del algoritmo en el grafo de dependencias, y por lo tanto se espera una mayor precisión en el resultado. Sin embargo, la técnica también puede ofrecer peores resultados, en especial cuando se utilizan pocos rayos.

3.1.2. Selección aleatoria de nodo incidente según la cantidad de nodos incidentes (LeSeEs$_{Sn}$)

Esta variante es equivalente a la variante anterior, pero el número aleatorio r del nodo i se selecciona entre 1 y $I(i) * Q$, donde $I(x)$ es la cantidad de nodos adyacentes al nodo x, y Q es una constante entre 0 y 1 seleccionada al inicio del algoritmo. La modificación introducida busca priorizar la búsqueda por el camino que, en el paso que se está efectuando la selección, ofrece mayor amplitud.

3.1.3. Múltiples ejecuciones aleatorias por nodo (LeSeEs$_{MS}$)

Las variantes anteriores tienen el inconveniente de elegir caminos aleatoriamente, lo cual puede resultar en caminos que no sean los más largos. Una solución posible es

computar desde un nodo varios caminos diferentes, de forma de incrementar la probabilidad de encontrar el camino más largo tomando ese nodo como el inicial. Otra forma de ver esta estrategia es como la introducción de más disparos o rayos dentro de una etapa de la heurística, es decir, se podría asimilar a utilizar una técnica de búsqueda local.

3.1.4. Método híbrido aleatorio-determinista (LeSeEs$_{Sh}$)

Otra forma de solucionar el problema anterior es notar que la cantidad de nodos incidentes disminuye al inspeccionar nodos con menor índice. Por esta razón, seleccionar los nodos incidentes de forma aleatoria durante todo el procedimiento puede tener una menor probabilidad de resultar en un camino óptimo. Por lo tanto, en esta variante se propone utilizar una selección aleatoria de nodos incidentes en la "mitad" superior de las filas de la matriz, y cambiar al método determinista original al llegar a la "mitad" inferior. De esta forma se busca aprovechar las ventajas de cada estrategia. El punto de corte no tiene porqué ser exactamente el medio de la matriz, de hecho puede ser encontrado experimentalmente para las distintas clases de matrices.

3.1.5. Uso de distribución normal (LeSeEs$_{Dn}$)

Otra variante con la cual se experimentó fue asumir que la profundidad de los nodos dentro de una matriz sigue una distribución similar a una normal o, por lo menos, aproximadamente simétrica, como se puede apreciar en el caso de la matriz *atmosmodd* en la Figura 14.

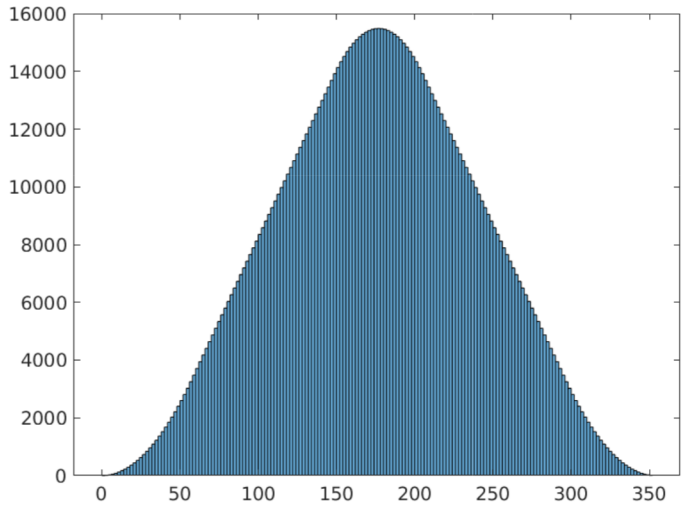


Figura 14: Distribución de las filas en los niveles para el triangulo inferior de la matriz atmosmodd.

Si esto se cumple, se puede asumir que el nodo de mayor profundidad tiene profundidad $2 * \eta$, donde η es el centro de la distribución normal. Para encontrar η, lo que se hace es asumir que la profundidad estimada por el método determinista es correcta, y tomando x filas al azar encontrar la profundidad máxima y mínima entre ellas, sumarla y dividirla entre 2. Esta variante tiene como característica principal que puede sobre-estimar el valor de los niveles.

3.2. Implementación en GPU

Otro aspecto que se abordó en el proyecto es el referido al desempeño computacional de la heurística. En este sentido se implementaron dos versiones distintas del algoritmo de rayos en GPU. La primera es una traducción directa del algoritmo original, empleando cada hilo de la GPU en un rayo distinto, mientras que la segunda emplea un mecanismo de reducción en un intento de mejorar los tiempos de cómputo en matrices con muchos niveles.

3.2.1. Traducción directa del algoritmo original

Debido a que la ejecución de cada nodo es independiente de las otras, el algoritmo es fácil de adaptar para GPUs, lo cual puede reducir su tiempo de ejecución de forma considerable. Además, luego de ejecutar el algoritmo en todos los r nodos es posible utilizar estrategias de reducción paralelas, como aquellas propuestas en [21], para obtener el nodo con mayor profundidad en tiempo logarítmico.

3.2.2. Variante logarítmica de la heurística

El algoritmo original utiliza un método determinista para seleccionar el siguiente nodo incidente a tomar, y éste es independiente de los nodos anteriores, por lo tanto es posible generar un vector $cols$ el cual contiene dicho nodo incidente. De no existir un nodo incidente se utiliza n como comodín. Este vector puede ser interpretado como un grafo bosque dirigido sin ciclos, donde los nodos sumidero son las filas en la matriz sin dependencias, y el algoritmo de rayos original devuelve el camino más largo en el mismo. En la GPU, debido a la enorme cantidad de hilos disponibles, es posible hacer esto en tiempo logarítmico respecto de n. Para lograr esto se crea otro vector $depths$ de largo n inicializado en 1, el cual representa el largo del camino más largo partiendo desde dicha fila hasta el nodo de la entrada en $cols$, o el camino más largo si dicha entrada es n. Una vez hecho esto, de forma paralela por cada fila, si la fila tiene en su entrada correspondiente del vector $cols$ un índice ind distinto de n se reemplaza esta entrada con el valor en $cols$ de ind, y se le suma a su entrada en $depths$ la entrada en $depths$ de ind (debido a las condiciones carrera en estas operaciones se recurre a la técnica de *double buffering*). Lo que se intenta hacer en este paso es acumular los caminos hallados hasta ese momento. Esto se realiza $log(n)$ veces y, una vez terminado, el vector $cols$ contiene solamente n, y por lo tanto el vector $depths$ contiene el largo del camino más largo partiendo desde cada fila. Utilizando una estrategia de reducción paralela como se mencionó anteriormente es posible extraer la entrada más alta de este vector, la cual se

corresponde con el camino más largo del grafo y el resultado del algoritmo. Esta variante del algoritmo de rayos devuelve los mismos resultados que el algoritmo original si se dispara un rayo por cada fila, y a diferencia de éste crece de forma logarítmica según el largo de n, en lugar de crecer en forma lineal según el largo del camino más largo encontrado. Este algoritmo se muestra en el Algoritmo 4 (ignorando las consideraciones de *double buffering*).

Otra forma de ver esta variante es considerar ecuaciones de niveles para cada fila (luego de la inicialización del vector *depths*) de la forma $niveles[i] = depths[i] + niveles[cols[i]]$ en el caso de las filas con dependencias, o $niveles[i] = depths[i]$ en el caso de las filas sin dependencias. Este último caso se generaliza como un caso particular del primero utilizando una fila "dummy" con índice n, como se explicó anteriormente. El algoritmo, entonces, reemplaza $niveles[cols[i]]$ con $depths[cols[i]] + niveles[cols[cols[i]]]$ en todas las filas a la vez, lo cual en la fila con mayor profundidad acorta el largo del camino más largo por una fila. Luego, esta operación se realiza de nuevo, lo cual acorta el largo de este camino por dos filas (una por el reemplazo de la dependencia en la fila con mayor profundidad, y otra por el reemplazo que se realizó en la iteración anterior en esta nueva dependencia). Por cada iteración del algoritmo, la cantidad de filas que se acortan en el camino se duplica, y se sabe que este camino (de largo la cantidad de niveles de la matriz) está acotado por n, por lo que realizando la operación $log(n)$ veces se garantiza que el camino se reduzca por completo. Por último, el largo del camino se recupera leyendo el vector *depths*, que acumuló las profundidades de los nodos en cada iteración.

Algoritmo 4 Variante logarítmica del algoritmo de rayos para estimar la cantidad de niveles de una matriz triangular superior dispersa almacenada en formato CSR

Input

$indices\_col$	Arreglo de índices de columnas de la matriz en formato CSR
ptr_fil	Arreglo de punteros a filas de la matriz en formato CSR
$cols$	Arreglo de dependencias de filas, inicializado según el criterio de la heurística original
$dephts$	Arreglo de profundidades de las filas, inicializado a 1
$proxima\_fila$	Índice de la fila a analizar

Output

$estimacion\_niveles$	Estimación de los niveles de la matriz

$proxima\_fila = cols[indice\_fila]$
for $i = 0$ **to** $log(n)$ **do**
  $cols[indice\_fila] = cols[proxima\_fila]$
  $depths[indice\_fila] += depths[proxima\_fila]$
  sincronizar
end for
$estimacion\_niveles = max(dephts)$
return $estimacion\_niveles$

3.3. Evaluación experimental de las heurísticas

Aquí se describen los detalles de la evaluación experimental realizada, cómo fue realizada, y los resultados de la misma.

3.3.1. Plataformas de hardware

Se utilizaron dos plataformas de hardware para la evaluación de los algoritmos presentados. En la Tabla 1 se describen las características de las mismas.

Tabla 1: Plataformas de hardware utilizadas en la propuesta de mejora de la heurística de estimación de niveles.

Plataforma	Plataforma 1	Plataforma 2
Procesador		
Modelo	Intel(R) Core(TM) i7-6700	Intel(R) Core(TM) i7-6700
Frecuencia	3.40 GHz	3.40 GHz
Tamaño de caché de primer nivel	32 KB	32 KB
Tamaño de caché de último nivel	8 MB	8 MB
Memoria		
Tamaño	65.69 GB	65.69 GB
Tarjeta gráfica		
Modelo	GeForce GTX 980 Ti	GeForce GTX 1060 3GB
Frecuencia del reloj	1000 MHz	1506 MHz
Núcleos CUDA	2816	1152
Tamaño de memoria	6 GB	3 GB
Ancho de banda de memoria	7.0 Gbps	8.0 Gbps
CUDA compute capability soportada	5.2	6.1

La diferencia principal entre ambas plataformas es la GPU. La GPU de la plataforma 1 tiene más del doble de núcleos CUDA, lo cual implica una mayor capacidad de paralelismo. Además, tiene el doble de memoria que la del sistema 2, lo cual permitiría almacenar matrices más grandes que la GPU del sistema 2. Sin embargo, esta última tiene una frecuencia de reloj 50 % más rápida, lo cual involucra una velocidad de cómputo mayor y un ancho de banda de memoria un poco mayor, lo que significa accesos a memoria más rápidos. Por último, la plataforma 2 tiene una versión de *CUDA (compute capability)* más alta, lo que podría permitirle ejecutar código más optimizado con

instrucciones más modernas.

3.3.2. Casos de estudio

Las matrices utilizadas para medir el desempeño del algoritmo propuesto fueron obtenidas de la SuiteSparse Matrix Collection [41], de la Universidad A&M de Texas. Se utilizaron las matrices comprendidas en la Tabla 2, a las cuales se les removieron los coeficientes ubicados por encima de la diagonal para convertirlas en matrices triangulares inferiores.

Tabla 2: Matrices de prueba utilizadas

Matriz	n	nnz	Niveles
Trefethen_20000	20,000	287,233	20,000
cage15	5,154,859	52,177,204	65
cnr-2000	325,557	1,788,943	181
co2010	201,062	688,349	27
com-DBLP	317,080	1,366,946	231
cont-300	180,895	629,694	2
crashbasis	160,000	1,232,812	600
dixmaanl	60,000	179,999	60,000
fe_tooth	78,136	530,727	24,095
filter3D	106,437	1,406,808	2,461
jan99jac120sc	41,374	199,885	3,347
linverse	11,999	53,988	11,999
mac_econ_fwd500	206,500	948,056	12,518
nemeth15	9,506	274,654	9,506
olm5000	5,000	12,498	5,000
pkustk14	151,926	7,494,215	1,075
preferentialAttachment	100,000	599,985	64
shyy161	76,480	228,482	13,038
smallworld	100,000	599,998	82,392
ss1	205,282	525,873	147

Las matrices fueron seleccionadas priorizando abarcar matrices de tamaños grandes y medianos, así como matrices con una gran cantidad de niveles.

3.3.3. Evaluación de la calidad de estimación

Este experimento consiste en comparar los resultados de todas las variantes de la heurística, así como la heurística original, en las matrices presentadas anteriormente.

En el caso de las variantes de LeSeEs$_{Sr}$, LeSeEs$_{MS}$ y LeSeEs$_{Sh}$ se seleccionó un R de 3, para LeSeEs$_{Sn}$ se seleccionó un Q de 0.25, y para LeSeEs$_{MS}$ se decidió usar 8 rayos

Tabla 3: Resultados experimentales, estimación de niveles por las diferentes variantes utilizando 1 % de rayos

Matriz	Niveles	Original	LeSeEs$_{Sr}$	LeSeEs$_{Sn}$	LeSeEs$_{MS}$	LeSeEs$_{Sh}$	LeSeEs$_{Dn}$
Trefethen...	20000	20000	8677	14359	8730	6475	20014
cage15.mtx	147	43	17	18	36	18	47
cnr-2000.mtx	181	64	36	49	15	35	139
co2010.mtx	27	13	10	10	11	13	17
com-DBLP.mtx	231	91	48	60	35	46	73
cont-300.mtx	2	2	2	2	2	2	3
crashbasis.mtx	600	600	503	599	512	599	603
dixmaanl.mtx	60000	60000	20001	20001	20002	60000	59983
fe_tooth.mtx	24095	878	533	1100	355	1069	1080
filter3D.mtx	2461	1481	700	1065	684	689	1454
jan99jac...	3347	2252	354	1293	346	253	2235
linverse.mtx	11999	11999	6060	9039	6099	10852	12002
mac_econ_f...	12518	2006	144	152	143	1503	2002
nemeth15.mtx	9506	9506	4801	7160	4844	2361	9505
olm5000.mtx	5000	5000	3379	4187	3407	5000	5004
pkustk14.mtx	1075	214	227	228	47	42	411
preferentialA...	64	50	24	49	26	37	62
shyy161.mtx	13038	160	160	160	160	160	623
smallworld.mtx	82392	24059	4454	35805	4648	45363	36606
ss1.mtx	147	145	143	143	145	145	142

por fila (analizando menos filas para mantener la misma cantidad de rayos que las otras variantes). Estos valores fueron determinados los más eficientes a la hora de encontrar niveles, mediante experimentos no formalizados.

Las Tablas 3 y 4 muestran la cantidad de niveles detectados utilizando una cantidad de hilos igual a 1 % y un 10 % de la cantidad de filas de la matriz respectivamente.

Tabla 4: Resultados experimentales, estimación de niveles por las diferentes variantes utilizando 10 % de rayos

Matriz	Niveles	Original	LeSeEs$_{Sr}$	LeSeEs$_{Sn}$	LeSeEs$_{MS}$	LeSeEs$_{Sh}$	LeSeEs$_{Dn}$
Trefethen...	20000	20000	8681	14357	8743	6470	19999
cage15.mtx	147	45	18	18	39	18	50
cnr-2000.mtx	181	75	44	55	15	43	149
co2010.mtx	27	17	14	14	13	18	21
com-DBLP.mtx	231	96	58	67	43	52	92
cont-300.mtx	2	2	2	2	2	2	3
crashbasis.mtx	600	600	503	599	516	599	601
dixmaanl.mtx	60000	60000	20001	20001	20002	60000	59996
fe_tooth.mtx	24095	1137	704	1149	405	1171	1137
filter3D.mtx	2461	1481	710	1070	710	734	1477
jan99jac...	3347	2252	358	1297	357	258	2251
linverse.mtx	11999	11999	6061	9037	6106	10842	12000
mac_econ_f...	12518	2006	191	199	172	1503	2004
nemeth15.mtx	9506	9506	4808	7162	4843	2374	9509
olm5000.mtx	5000	5000	3377	4194	3407	5000	5002
pkustk14.mtx	1075	240	227	228	49	42	434
preferentialA...	64	53	28	49	29	40	53
shyy161.mtx	13038	160	160	160	160	160	635
smallworld.mtx	82392	27493	4870	35956	5009	47595	36610
ss1.mtx	147	145	143	143	145	145	143

A partir de estos datos se puede apreciar que el uso de una cantidad de ejecuciones igual a 10 % de n no aumenta la precisión de forma significativa en ninguna de las variantes presentadas frente a usar 1 % de n. Además, se concluye que la heurística original y la variante de distribución normal son las que alcanzan los mejores resultados. Profundizando en estos resultados se pueden definir cuatro categorías para los resultados, según el porcentaje del valor real alcanzado por la estimación:

1. A (Estimación igual o muy similar al valor real.)
2. B (Estimación mayor al 50 % del valor real.)
3. C (Estimación mayor al 25 % del valor real.)
4. D (Estimación menor al 25 % del valor real.)

A partir de estas categorías, se clasificaron las variantes de la heurística, así como la heurística original, en la Tabla 5.

Tabla 5: Clasificación de las estimaciones de la heurística original y sus variantes.

Solver	A	B	C	D
Original	8	3	5	4
LeSeEs$_{Sr}$	2	4	5	9
LeSeEs$_{Sn}$	3	4	7	6
LeSeEs$_{MS}$	2	4	5	9
LeSeEs$_{Sh}$	5	3	3	9
LeSeEs$_{Dn}$	9	4	4	3

3.3.4. Evaluación del desempeño

Se ejecutó la heurística de predicción de niveles en GPU en cada matriz, utilizando el 1 % del n de cada matriz como la cantidad de filas iniciales analizadas, así como la variante logarítmica presentada, y se graficaron los resultados en la Figura 15 ordenando las matrices según la cantidad de niveles y transformando los ejes con una función logarítmica para una mejor interpretación de los datos.

Como puede verse, la traducción directa en GPU tiene un mejor rendimiento en matrices con pocos niveles que la variante logarítmica, pero el tiempo de ejecución se eleva rápidamente en la primera al aumentar la cantidad de niveles, mientras que se mantiene constante en la última.

3.3.5. Evaluación de la predicción (utilizando el dataset original)

En la Tabla 6 se muestran los resultados obtenidos de la ejecución del algoritmo propuesto en las plataformas de hardware y matrices mencionadas anteriormente. Los tiempos presentados fueron el promedio de 5 ejecuciones independientes, y fueron divididos sobre el tiempo de ejecución en CPU en el sistema correspondiente y promediados. La cantidad de disparos es el número divisible entre 1024 más pequeño que sea mayor a la cantidad de rayos lanzada (exceptuando el caso donde se lanza solo un rayo), dado que el algoritmo trabaja a nivel de bloque de GPU, donde cada bloque tiene 1024 threads. Los porcentajes indican la cantidad de rayos en comparación a la cantidad de filas de una matriz (por ejemplo si una matriz tiene 2.000.000 filas, la entrada 50 % representaría 1.000.000 rayos, e involucraría el siguiente múltiplo de 1024, o sea 1.000.448 rayos).

3.4. Revisión de estudios previos de predicción de solvers mediante aprendizaje automático

Se replicaron los experimentos de [12], pero debido a un error se omitió la característica de precisión aritmética de punto flotante, y se notó que con este cambio la precisión fue ligeramente superior a la presentada en dicho artículo. Además, la precisión en modelos con y sin niveles fueron muy similares, con el modelo que no utiliza niveles siendo apenas superior. Estos resultados pueden verse en la Tabla 7. En esta tabla se incluye el error

Figura 15: Tiempos de ejecución (en ms) del algoritmo de rayos en GPU y la variante logarítmica según n, con ambos ejes transformados con una función logarítmica.

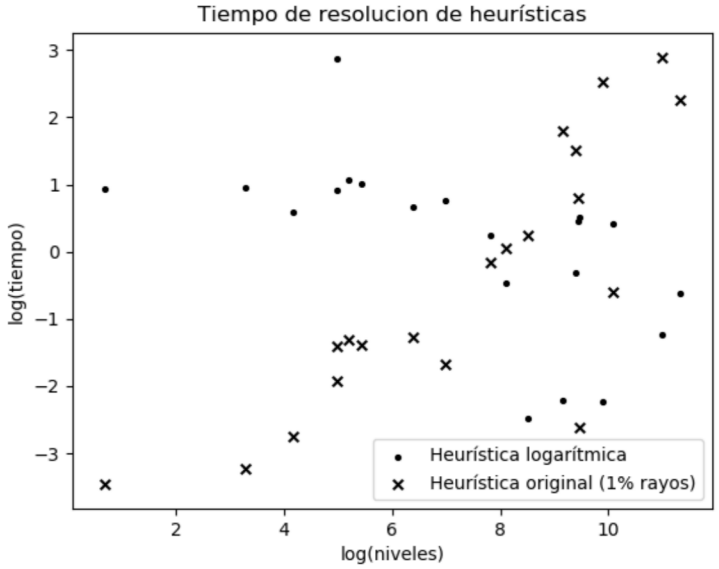


Tabla 6: Resultados experimentales, velocidad de ejecución de las variantes de GPU de la heurística sobre la velocidad de ejecución de la heurística original en CPU.

	Traducción directa en GPU		Variante logarítmica	
Cantidad de ejecuciones	Sistema 1	Sistema 2	Sistema 1	Sistema 2
1	67.07 %	310.35 %	3050.49 %	1350.57 %
1 %	32.75 %	28.72 %	1437.08 %	812.60 %
2 %	16.96 %	13.82 %	727.23 %	408.27 %
3 %	12.28 %	11.04 %	502.22 %	287.12 %
4 %	10.05 %	8.80 %	379.43 %	220.18 %
5 %	8.95 %	7.85 %	332.00 %	193.24 %
6 %	7.95 %	6.95 %	273.41 %	160.26 %
7 %	6.79 %	6.12 %	231.09 %	137.06 %
8 %	6.33 %	5.72 %	205.77 %	119.50 %
9 %	5.73 %	5.04 %	191.30 %	111.41 %
10 %	5.49 %	4.80 %	172.29 %	100.21 %
25 %	3.22 %	3.14 %	67.83 %	39.92 %
50 %	2.23 %	2.41 %	35.83 %	21.95 %
100 %	1.89 %	2.32 %	29.95 %	18.83 %

de tiempo de ejecución promedio (ARE por su sigla en inglés), el cual se computa de la siguiente manera:

$$ARE = \frac{1}{d}\sum_{1}^{d}\frac{SM_{runtime}(i) - BM_{runtime}(i)}{BM_{runtime}(i)}$$

donde $SM_{runtime}$ es el tiempo de ejecución que toma el método seleccionado, $BM_{runtime}$ es el menor tiempo de ejecución entre los métodos analizados, y d es el número de instancias en el conjunto de datos. Esta métrica ayuda a entender que tan "precisa" es la predicción desde el punto de vista del problema, ya que penaliza más severamente predicciones incorrectas con errores grandes que predicciones incorrectas con errores pequeños.

Tabla 7: Resultados de replicar los experimentos de aprendizaje automático presentados en [12], excluyendo la característica de precisión aritmética (*prec*).

Características excluidas	Modelo	Precisión	ARE
—	WKNN	0.82	1.00
	Decision tree	0.80	1.03
	Bagged tree	0.82	1.01
$locality, locality_{mr}$	WKNN	0.82	1.02
	Decision tree	0.79	1.03
	Bagged tree	0.82	1.04
$locality, locality_{mr}, niveles$	WKNN	0.81	1.01
	Decision tree	0.77	1.06
	Bagged tree	0.81	1.01
$locality, locality_{mr}, niveles, bw_{avg}, nnz_{max}, nnz_{stdev}$	WKNN	0.80	1.02
	Decision tree	0.65	1.22
	Bagged tree	0.80	1.05

Estos resultados implican que nnz y n serían suficientes para decidir el *solver* óptimo, y por lo tanto no serían importantes. Este tema es estudiado con mayor profundidad en la siguiente sección.

4. Estudio de features y generación de datasets

En el capítulo anterior se obtuvieron resultados que ponen en duda la utilidad de los niveles para seleccionar el *solver* óptimo, y por lo tanto la utilidad del estimador de niveles. En teoría, la cantidad de niveles de una matriz impacta en el potencial de una matriz de ser procesada en paralelo, en especial en algoritmos como los de resolución por niveles presentados anteriormente, donde esta relación es explícita en el diseño de los mismos. Una hipótesis que explica esta falta de impacto de la cantidad de niveles en el análisis es que la cantidad de matrices utilizadas para el entrenamiento fue demasiado limitada, lo cual puede generar artefactos estadísticos en las características de estas matrices que no reflejen la realidad, y que con un conjunto de pruebas de mayor tamaño esta variable ganaría mayor importancia. En esta sección se extiende el análisis para estudiar este aspecto, utilizando un conjunto de datos de mayor dimensión.

Para esto, la propuesta consiste en generar diversas matrices de forma aleatoria, con varios parámetros, y ejecutar los *solvers* a estudiar en las mismas, registrando los tiempos de ejecución de cada uno. Luego, estos tiempos son utilizados como entrada a diversos algoritmos de aprendizaje automático supervisado, para intentar clasificar las matrices, según sus parámetros, en los *solvers* que las resuelven de manera más rápida.

Como se mencionó anteriormente, se utilizó la suite Classification App provista por MATLAB como plataforma para llevar a cabo tanto el entrenamiento de los modelos de aprendizaje automático como la evaluación de los mismos.

4.1. Plataforma de hardware

Se utilizó la plataforma de hardware presentada en la Tabla 8 para la evaluación de los tiempos de los solvers estudiados. La latencia del acceso a disco y de transferencias no es importante, ya que los tiempos fueron medidos una vez la matriz se encontrase ya almacenada en la memoria de la GPU.

4.2. Utilización de datasets aumentados para estudiar el impacto de la cantidad de niveles

Debido a la limitada cantidad de matrices reales disponibles para descargar, se decidió generar matrices dispersas mediante un algoritmo aleatorio. Esto permite la creación de una cantidad arbitraria de matrices únicas, con tamaño, cantidad de entradas no nulas y cantidad de niveles predefinidas. Una desventaja de este método es la posible aparición de *bias* en las matrices, es decir, de artefactos probabilísticos generados por el diseño del algoritmo generador.

El algoritmo en cuestión está pensado en el contexto de la solución de sistemas lineales dispersos triangulares. En particular, funciona generando un camino aleatorio con largo igual a la cantidad de niveles en una matriz del tamaño seleccionado, y luego rellenando la matriz, evitando introducir no ceros en las entradas que se encuentren en los triángulos definidos por dos nodos contiguos del camino definido previamente y la entrada que se encuentre en la columna el primer nodo y la fila del segundo.

Tabla 8: Plataforma de hardware utilizada para la propuesta de clasificación de matrices mediante aprendizaje automático.

Procesador	
Modelo	Intel(R) Core(TM) i7-6700
Frecuencia	3.40 GHz
Tamaño de caché de primer nivel	32 KB
Tamaño de caché de último nivel	8 MB
Memoria	
Tamaño	65.69 GB
Targeta gráfica	
Modelo	GeForce GTX 1060 3GB
Frecuencia del reloj	1506 MHz
Núcleos CUDA	1152
Tamaño de memoria	3 GB
Ancho de banda de memoria	8.0 Gbps
CUDA compute capability soportada	6.1

Debido a la forma en la que se construyen las matrices la heurística siempre detecta la cantidad de niveles exacta, cosa que no pasa en general, por lo que éstas no son aptas para ser utilizadas en estudios relacionados a la heurística de estimación de niveles (ver sección 3).

Para este estudio se analizaron tres tamaños de matrices:

1. Pequeña ($n = 100{,}000$)
2. Mediana ($n = 1{,}000{,}000$)
3. Grande ($n = 10{,}000{,}000$)

El tamaño, o dimensión, de una matriz (medido en cantidad de filas, o n) es importante porque se relaciona fuertemente con el número de hilos que pueden estar trabajando concurrentemente sobre la misma.

Además, se analizaron tres niveles de dispersión de matriz:

1. Muy dispersa ($nnz = 400{,}000$ en matrices pequeñas, $nnz = 1{,}000{,}000$ en matrices medianas y $nnz = 10{,}000{,}000$ en matrices grandes)
2. Medianamente dispersa ($nnz = 4{,}000{,}000$ en matrices pequeñas, $nnz = 50{,}000{,}000$ en matrices medianas y $nnz = 150{,}000{,}000$ en matrices grandes)
3. Poco dispersa ($nnz = 40{,}000{,}000$ en matrices pequeñas y $nnz = 150{,}000{,}000$ en matrices medianas. No hay matrices grandes, por razones explicadas más adelante)

El nivel de dispersión de una matriz (medido en cantidad de elementos no nulos, o nnz) se relaciona con la cantidad de procesamiento que debe realizarse sobre una matriz, ya que por cada uno se requiere realizar una multiplicación y una suma al resolver sistemas lineales.

Por último, se analizó la cantidad de niveles de la matriz:

1. Pocos niveles ($niveles = n * 0,01$)
2. Cantidad media de niveles ($niveles = n * 0,4$)
3. Muchos niveles ($niveles = n * 0,8$)

Estas cantidades (incluyendo el nivel de dispersión en matrices de cada tamaño) fueron obtenidas analizando matrices encontradas en SuiteSparse Matrix Collection [41], de la Universidad A&M de Texas.

Combinando estas categorías, que son además las que se analizarían en el modelo de aprendizaje automático, se llega a 27 grupos distintos de matrices. Sin embargo, las matrices grandes y pocos dispersas no fueron probadas, debido a que el gran tamaño de las mismas en memoria causa que no sea posible manipularlas en el hardware de prueba de la misma forma que las otras matrices, llegando a 24 grupos de prueba finales. Para cada uno de estos grupos se generaron 100 matrices aleatorias distintas.

4.3. Estudio de características matriciales relevantes

Se ejecutó cada uno de los *solvers*: el provisto por *cuSparse* (Sptrsv_{CS}), dos variantes de la resolución por niveles (Sptrsv_{L1} y Sptrsv_{L2}), el algoritmo asincrónico base (Sptrsv_{AB}) y sus variantes de valor inicial de NaN (Sptrsv_{AN}), reordenamiento de filas (Sptrsv_{AR}) y con procesamiento de múltiples filas (Sptrsv_{AM}) en las matrices generadas, una vez cada una, y se llegó a los siguientes resultados:

1. Solamente dos de los algoritmos (Sptrsv_{AN} y Sptrsv_{AM}) fueron superiores en al menos una matriz, con Sptrsv_{AM} siendo superior en 2257 matrices, es decir, un 94 % de las matrices generadas.
2. Si se dividen las matrices según el conjunto se obtiene la Tabla 9.
3. La máxima ganancia absoluta de tiempo en una matriz por usar el algoritmo Sptrsv_{AN} sobre la alternativa superior en la mayoría de los casos es de 3.39 segundos, y la mayor ganancia porcentual es de 14 %.

Estos resultados son consistentes con estudios previos, que indicaban que Sptrsv_{AM} es el algoritmo superior en la mayoría de los casos. Las posibles ganancias de tiempo obtenidas por usar Sptrsv_{AN} son limitadas, y serían menores en muchos casos al tiempo requerido para ejecutar un algoritmo que decida cuál *solver* utilizar, sin embargo en escenarios donde múltiples sistemas lineales con los mismos parámetros sean resueltos en secuencia, estas ganancias de tiempo podrían ser notables.

Tabla 9: Porcentaje de las matrices generadas en las cuales el algoritmo asincrónico con procesamiento en múltiples filas (sptrsv$_{AM}$) es el *solver* superior, separado por grupo de matrices.

Pocos niveles			
Tamaño	Pequeña	Mediana	Grande
Muy dispersa	100 %	100 %	100 %
Medianamente dispersa	90 %	100 %	100 %
Poco dispersa	67 %	0 %	—

Cantidad media de niveles			
Tamaño	Pequeña	Mediana	Grande
Muy dispersa	100 %	100 %	100 %
Medianamente dispersa	100 %	100 %	100 %
Poco dispersa	100 %	100 %	—

Muchos niveles			
Tamaño	Pequeña	Mediana	Grande
Muy dispersa	100 %	100 %	100 %
Medianamente dispersa	100 %	100 %	100 %
Poco dispersa	100 %	100 %	—

Los resultados de cada matriz fueron recopilados junto con su número de niveles, número de filas y número de entradas no nulas, y fueron utilizados como entrada para todos los modelos de la *Classification Learner App*. Estos modelos fueron verificados utilizando *5-fold cross validation*. En la Tabla 10 se puede apreciar los resultados de algunos de los modelos luego del entrenamiento.

Además, las matrices de confusión (las tablas que describen la precisión del modelo, así como los falsos positivos/negativos) de los *solvers* es la presentada en la Figura 16 para modelos sin niveles y en la Figura 17 para modelos con niveles.

Como puede verse, los algoritmos que utilizan este conjunto de datos necesitan la cantidad de niveles para alcanzar precisiones superiores a la solución trivial de seleccionar siempre el *solver* más eficiente para la mayoría de las matrices, lo cual es una solución no óptima para matrices pequeñas con pocos niveles con este conjunto de *solvers*.

Figura 16: Matriz de confusión de los modelos óptimos que no utilizan niveles para el problema de selección de *solver*. Como puede verse, el modelo simplemente elige el *solver* con procesamiento de múltiples filas (SPTRSV_{AM}) para todas las matrices, el cual fue superior en la mayoría de los sistemas del conjunto de datos.

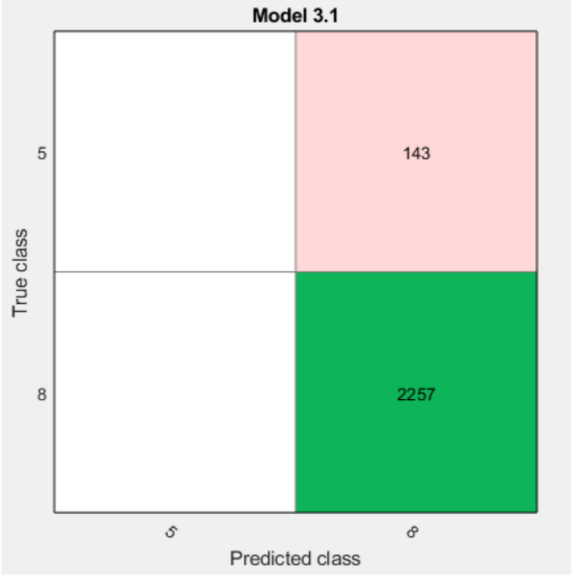


Figura 17: Matriz de confusión de los modelos óptimos que utilizan niveles para el problema de selección de *solver*.

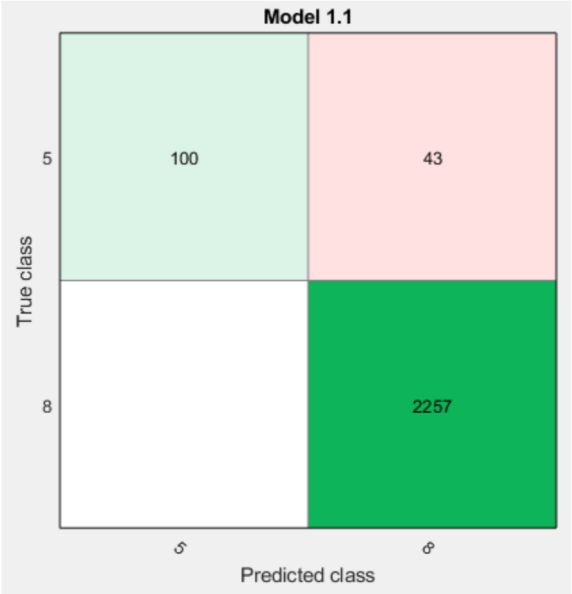


Tabla 10: Resultados de aplicar todos los modelos de aprendizaje automático de la *Classification Learner App* al conjunto de datos generados, tanto con y sin la utilización de niveles. En negrita se muestran las precisiones más altas de ambas.

Categoría	Modelo	Precisión sin niveles	Precisión con niveles
Decision trees	Fine tree	**0.940**	**0.982**
	Medium tree	**0.940**	**0.982**
	Coarse tree	**0.940**	**0.982**
Discriminant	Lin. disc.	**0.940**	0.940
	Quad. disc.	0.801	(falló)
Naive Bayes	Gaussian	0.899	0.810
	Kernel	**0.940**	0.940
Logistic regressions	Log. regr.	**0.940**	**0.982**
SVM	Linear SVM	**0.940**	**0.982**
	Quadratic	**0.940**	**0.982**
	Cubic Gauss.	**0.940**	**0.982**
	Fine Gauss.	**0.940**	**0.982**
	Medium Gauss.	**0.940**	**0.982**
	Coarse Gauss.	**0.940**	0.940
KNN	Fine	0.899	**0.982**
	Medium	0.899	**0.982**
	Coarse	0.899	**0.982**
	Cosine	0.899	**0.982**
	Cubic	0.899	**0.982**
	Weighted	0.899	**0.982**
Ensemble	Boosted trees	**0.940**	**0.982**
	Bagged trees	**0.940**	**0.982**
	Subspace disc.	**0.940**	0.940
	Subspace KNN	0.803	**0.982**
	RUSBoost. trees	0.801	0.968

5. Conclusiones y trabajo futuro

En esta sección se discuten los resultados obtenidos en las Secciones 3 y 4, así como las conclusiones extraídas de los mismos. También se presentan propuestas de trabajo futuro para profundizar más en los temas presentados en este proyecto.

5.1. Conclusiones

En los últimos años se han creado múltiples algoritmos para resolver sistemas lineales dispersos triangulares, intentando reducir los tiempos de resolución en la mayor cantidad de matrices posibles. Esto ha introducido una preocupación por determinar la selección óptima de algoritmo de resolución para cada matriz.

En este proyecto se han extendido diversos trabajos previos. Primero, se continuó el desarrollo de la heurística de estimación de niveles, proponiendo nuevas variantes para aumentar su precisión sin afectar su costo computacional. También se desarrollaron variantes en GPUs para aumentar la velocidad de las estimaciones. Segundo, se repitió la investigación de selección de *solver* utilizando un conjunto de datos más amplio generado automáticamente.

Los resultados sobre un conjunto de 20 matrices dispersas de la colección SuiteSparse, mostraron que la heurística original, así como su variante basada en la distribución normal (LeSeEs_{Dn}) son las que presentan mayores niveles de precisión. En cuanto al desarrollo en GPU, se lograron importantes aceleraciones en las variantes implementadas, presentando tiempos de ejecución suficientemente cortos para hacer viable su uso en la selección de *solvers*. Las variantes propuestas a la heurística y los resultados obtenidos se publicaron en [19]. Sobre estos conjuntos de datos parece no ser determinante el número de niveles.

Por último, utilizando conjuntos de datos generados automáticamente, con matrices en distintas categorías, se llegó a la conclusión que los niveles tienen una influencia menor pero significativa a la hora de seleccionar el solver óptimo, lo cual hace que los niveles sean una característica útil de encontrar en situaciones donde el mismo patrón de no ceros sea utilizado en múltiples sistemas.

5.2. Trabajo futuro

Como trabajo futuro, se propone estudiar la eficacia de la estimación de la heurística de predicción de niveles como reemplazo del valor exacto en modelos de aprendizaje automático. Además, se sugiere la utilización de una mayor cantidad de plataformas de hardware para generalizar los resultados a una mayor cantidad de sistemas.

También se propone continuar la investigación sobre selección de *solvers* mediante aprendizaje automático, optimizando las características utilizadas, eliminando características poco útiles y agregando nuevas, buscar algoritmos de estimación para características difíciles de calcular, y mejorando la generación de instancias de prueba, intentando cubrir más terreno con las mismas en el espacio de matrices.

En cuanto a la heurística de estimación de niveles, se propone continuar su mejora, tanto intentando mejorar su precisión, estudiando y posteriormente minimizando las principales causas de error en la estimación, y su desempeño computacional, concentrándose especialmente en el uso eficiente de las capacidades de paralelismo de la GPU.

Finalmente, se propone la creación de nuevos algoritmos de resolución de sistemas lineales dispersos, que se concentren fuertemente en "nichos" del espacio de matrices (matrices muy dispersas, o matrices con más de 1.000.000 filas, por ejemplo), ya que utilizando mecanismos similares a los propuestos en este proyecto sería muy fácil utilizar un modelo de aprendizaje automático para seleccionar entre varios algoritmos que cubran diferentes "nichos", según las características de la matriz del sistema a resolver.

6. Referencias

[1] Edward Anderson y Youcef Saad. "Solving sparse triangular linear systems on parallel computers". En: *International Journal of High Speed Computing* 01.01 (1989), págs. 73-95. DOI: 10.1142/S0129053389000056. eprint: https://doi.org/10.1142/S0129053389000056.

[2] Hartwig Anzt, Edmond Chow y Jack Dongarra. "Iterative Sparse Triangular Solves for Preconditioning". En: *Euro-Par 2015: Parallel Processing*. Ed. por Jesper Larsson Träff, Sascha Hunold y Francesco Versaci. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, págs. 650-661. ISBN: 978-3-662-48096-0.

[3] Hartwig Anzt, Stanimire Tomov y Jack Dongarra. "Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C-σ formats on NVIDIA GPUs". En: *University of Tennessee, Tech. Rep. ut-eecs-14-727* (2014).

[4] N. Bell y M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". En: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Nov. de 2009, págs. 1-11. DOI: 10.1145/1654059.1654078.

[5] Leo Breiman. "Bagging predictors". En: *Machine learning* 24.2 (1996), págs. 123-140.

[6] Aydin Buluç y col. "Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks". En: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: ACM, 2009, págs. 233-244. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584053.

[7] Ali Cevahir, Akira Nukada y Satoshi Matsuoka. "Fast Conjugate Gradients with Multiple GPUs". En: vol. 2009. Ene. de 2009, págs. 893-903. DOI: 10.1007/978-3-642-01970-8_90.

[8] *CUDA Zone — NVIDIA Developer*. URL: https://developer.nvidia.com/cuda-zone. (accessed: 7/12/2020).

[9] S. A. Dudani. "The Distance-Weighted k-Nearest-Neighbor Rule". En: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-6.4 (1976), págs. 325-327.

[10] E. Dufrechou y P. Ezzatti. "Solving Sparse Triangular Linear Systems in Modern GPUs: A Synchronization-Free Algorithm". En: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Mar. de 2018, págs. 196-203. DOI: 10.1109/PDP2018.2018.00034.

[11] E. Dufrechou y P. Ezzatti. "Using analysis information in the synchronization-free GPU solution of sparse triangular systems". En: *Concurrency and Computation: Practice and Experience* 32.10 (2020). e5499 cpe.5499, e5499. DOI: 10.1002/cpe.5499.

[12] E. Dufrechou, P. Ezzatti y E. S. Quintana-Orti. "Automatic Selection of Sparse Triangular Linear System Solvers on GPUs through Machine Learning Techniques". En: *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2019, págs. 41-47.

[13] R. Eberhardt y M. Hoemmen. "Optimization of Block Sparse Matrix-Vector Multiplication on Shared-Memory Parallel Architectures". En: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, págs. 663-672.

[14] Daniel Erguiz, Ernesto Dufrechou y Pablo Ezzatti. "Assessing Sparse Triangular Linear System Solvers on GPUs". En: oct. de 2017, págs. 37-42. DOI: 10.1109/SBAC-PADW.2017.15.

[15] Liubov A. Flores y col. "CT Image Reconstruction Based on GPUs". En: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, págs. 1412-1420. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.05.308.

[16] N. Galoppo y col. "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware". En: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 2005, págs. 3-3. DOI: 10.1109/SC.2005.42.

[17] Alan George y col. "Solution of sparse positive definite systems on a shared-memory multiprocessor". En: *International Journal of Parallel Programming* 15.4 (ago. de 1986), págs. 309-325. ISSN: 1573-7640. DOI: 10.1007/BF01407878. URL: https://doi.org/10.1007/BF01407878.

[18] Federico González Madina. "Análisis de aerogeneradores de eje vertical para entornos urbanos. Modelación numérica 2D de rotores Savonius." En: (2020). Tesis de maestría. Universidad de la República (Uruguay). Facultad de Ingeniería. URL: https://hdl.handle.net/20.500.12008/24988.

[19] Eduardo González, Ernesto Dufrechou y Pablo Ezzatti. "Estimating the parallelism in the solution of sparse triangular linear systems". En: *2020 39th International Conference of the Chilean Computer Science Society (SCCC)*. 2020, págs. 1-8. DOI: 10.1109/SCCC51225.2020.9281169.

[20] Mark Harris. "Fast Fluid Dynamics Simulation on the GPU". En: *Fluid Dynamics* 1 (ene. de 2005), págs. 637-666. DOI: 10.1145/1198555.1198790.

[21] Mark Harris. *Optimizing Parallel Reduction in CUDA*. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. (accessed: 25/3/2020).

[22] *HB Files - The Harwell Boeing Sparse Matrix File Format*. URL: https://people.sc.fsu.edu/~jburkardt/data/hb/hb.html. (accessed: 6/11/2020).

[23] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2002. ISBN: 9781410612403.

[24] Geoffrey E Hinton y col. "Improving neural networks by preventing co-adaptation of feature detectors". En: *arXiv preprint arXiv:1207.0580* (2012).

[25] A. B. Kahn. "Topological Sorting of Large Networks". En: *Commun. ACM* 5.11 (nov. de 1962), págs. 558-562. ISSN: 0001-0782. DOI: 10.1145/368996.369025. URL: https://doi.org/10.1145/368996.369025.

[26] D.B. Kirk y W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2016. ISBN: 9780128119877.

[27] Weifeng Liu y col. "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves". En: *Euro-Par 2016: Parallel Processing*. Ed. por Pierre-François Dutot y Denis Trystram. Cham: Springer International Publishing, 2016, págs. 617-630. ISBN: 978-3-319-43659-3.

[28] Raul Marichal, Ernesto Dufrechou y Pablo Ezzatti. "Towards a Lightweight Method to Predict the Performance of Sparse Triangular Solvers on Heterogeneous Hardware Platforms". En: feb. de 2020, págs. 109-121. ISBN: 978-3-030-41004-9. DOI: 10.1007/978-3-030-41005-6_8.

[29] *Matrix Market: File Formats*. URL: https://math.nist.gov/MatrixMarket/formats.html. (accessed: 6/11/2020).

[30] Maxim Naumov. "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU". En: *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).

[31] A. Navada y col. "Overview of use of decision tree algorithms in machine learning". En: *2011 IEEE Control and System Graduate Research Colloquium*. 2011, págs. 37-42. DOI: 10.1109/ICSGRC.2011.5991826.

[32] *New GPU-accelerated Weather Forecasting System Dramatically Improves Accuracy – NVIDIA Developer News Center*. URL: https://news.developer.nvidia.com/new-gpu-accelerated-weather-forecasting-system-dramatically-improves-accuracy/. (accessed: 7/12/2020).

[33] R. A. Nugrahaeni y K. Mutijarsa. "Comparative analysis of machine learning KNN, SVM, and random forests algorithm for facial expression classification". En: *2016 International Seminar on Application for Technology of Information and Communication (ISemantic)*. 2016, págs. 163-168. DOI: 10.1109/ISEMANTIC.2016.7873831.

[34] *OpenFOAM*. URL: https://www.openfoam.com/. (accessed: 26/4/2021).

[35] Payam Refaeilzadeh, Lei Tang y Huan Liu. "Cross-Validation". En: (2009). Ed. por Ling Liu y M. Tamer Özsu, págs. 532-538. DOI: 10.1007/978-0-387-39940-9_565.

[36] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." En: *Psychological review* 65.6 (1958), pág. 386.

[37] Arthur L Samuel. "Some studies in machine learning using the game of checkers". En: *IBM Journal of research and development* 3.3 (1959), págs. 210-229.

[38] S. Soller. *GPGPU Origins and GPU Hardware Architecture*. Hochschule der Medien, 2011. URL: https://books.google.com.uy/books?id=rNSdwgEACAAJ.

[39] R. Srinivasan. *XDR: External Data Representation Standard*. RFC 1832. RFC Editor, ago. de 1995.

[40] D. Steinkraus, I. Buck y P. Y. Simard. "Using GPUs for machine learning algorithms". En: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. 2005, 1115-1120 Vol. 2. DOI: 10.1109/ICDAR.2005.251.

[41] *SuiteSparse Matrix Collection*. URL: https://sparse.tamu.edu/. (accessed: 20/5/2021).

[42] *Train models to classify data using supervised machine learning - MATLAB*. URL: https://www.mathworks.com/help/stats/classificationlearner-app.html. (accessed: 7/2/2021).