Proyecto: Implementación de matrices dispersas usando listas doblemente ligadas y soporte de multiplicación entre matrices y escalar-matriz, suma entre matrices.

CINVESTAV - Departamento de computación

Autor: André Fabián Castellanos Aldama

3 de Diciembre del 2021

El código lo puede encontrar en el siguiente link

Objetivo

1. Representar adecuadamente las matrices con listas doblemente ligadas entre filas y columnas.
2. Implementar inserción de valores en lugar.
3. Implementar suma entre matrices.
4. Implementar multiplicación entre matrices y entre una matriz y un escalar.
5. Mostrar resultados de una correcta implementación.

Representación de matrices dispersas con listas doblemente ligadas.

En vez de representar que los símbolos <> representen un arreglo, en este contexto será una lista doblemente ligada, donde sus elementos están separadas por comas y un par algebraico se representa como (). Para las matrices dispersas y tratando de seguir una representación acorde a lo pedido en clase, se propone lo siguiente:

Cada fila tendrá su valor y una lista de sus valores dentro de la misma que tendrán su valor de columna y su valor. Si un elemento de la matriz es 0, entonces no aparece en ninguna lista. Ejemplo, sea la siguiente matriz dispersa con dimension 4x4:

$$
\begin{gathered}
< (fila = 0, < (columna = 1, valor = 2), (columna = 2, valor = 8) >), \\
(fila = 1, < (columna = 0, valor = 3) >), \\
(fila = 3, < (columna = 0, valor = 1), (columna = 1, valor = 7) >) > \\
= \begin{pmatrix} 0 & 2 & 8 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 7 & 0 & 0 \end{pmatrix}
\end{gathered}
$$

Implementación de la representación junto con prototipos de funciones a usar más adelante

La clase quedaría como sigue:

```
class Matriz {
  unsigned int _fila_dim, _columna_dim;
  std::list<std::pair<unsigned int,std::list<std::pair<unsigned int,int>>>> _matriz;
  int GetValue(unsigned int fila, unsigned int columna) const;
 public:
  Matriz(unsigned int fila_dim, unsigned int columna_dim);
  void Print();
  void PrintAsMatriz();
  void InsertElement(unsigned int fila, unsigned int columna, int valor);
  friend Matriz operator+(Matriz const &matriz1, Matriz const &matriz2);
  friend Matriz operator*(Matriz const &matriz1, Matriz const &matriz2);
  friend Matriz operator*(int const &value, Matriz const &matriz);
};
```

Cada matriz tendrá las dimensiones de su fila y su columna, la representación de una matriz dispersa como se explico en la sección anterior. También tenemos su constructor que pide sus dimensiones, se incluye un método para imprimir en dos formatos con la representación y como una matriz normal. De ahí, se incluyen los métodos requeridos junto con el método obtener valor que será de utilidad en la multiplicación.

Inserción de un valor dado su fila y columna

Tenemos el prototipo `void InsertElement(unsigned int fila, unsigned int columna, int valor)` que pide la fila, columna y valor que se insertará. Si esa fila y columna existen, entonces sobreescribiremos el valor. En este método debe verificarse que se cumple con que la fila y la columna estén dentro del rango de fila o columna.

Procedimiento:

1. Buscamos la fila
2. Si no la encontramos y está vacía la matriz entonces solo lo insertamos como va.
3. Si no encontramos la fila que se busca entonces insertaremos la fila en donde tenga una fila menor atras y una mayor adelante.
4. Si lo encontramos, entonces buscamos dentro de la lista de columnas si existe la columna.
5. Si existe la columna, entonces solo reescribimos el valor.
6. Si no existe la columna, entonces insertamos la fila en donde tenga una columna menor detrás y una mayor adelante.

El código quedaría como sigue:

```
void Matriz::InsertElement(unsigned int fila, unsigned int columna, int valor) {

  /*Si los valores no están en donde deben estar entonces no se inserta*/
  if (!(0 <= fila < this->_fila_dim && 0 <= columna < this->_columna_dim && valor != 0))
    return;

  /*Buscar fila*/
  auto matrizIter = std::find_if(this->_matriz.begin(), this->_matriz.end(),
   [&fila](const std::pair<unsigned int, std::list<std::pair<unsigned int, int>>> &el) {
     if (el.first == fila) {
       return true;
     } else {
       return false;
     }
   });

  /*Si no está la fila*/
  if (matrizIter == this->_matriz.end()) {
    /*Si está vacio insertar así nomás*/
    if (this->_matriz.empty()) {
      std::list<std::pair<unsigned int, int>> lista_fila;
      lista_fila.emplace_back(columna, valor);
      this->_matriz.emplace_back(fila, lista_fila);
    } else {
      /*Si no está vacio entonces buscar después de que fila va*/
      auto matrizIterFila = this->_matriz.begin();

      /*Buscamos el apuntador donde la fila sea menor al siguiente*/
      while (matrizIterFila != this->_matriz.end()) {
        if (matrizIterFila->first < fila)
          ++matrizIterFila;
        else
          break;
      }

      /*Insertamos la fila*/
      std::list<std::pair<unsigned int, int>> lista_fila;
      lista_fila.emplace_back(columna, valor);
      std::pair<unsigned int, std::list<std::pair<unsigned int, int>>> fila_pair;
      fila_pair.first = fila;
      fila_pair.second = lista_fila;
      this->_matriz.insert(matrizIterFila, fila_pair);
    }
  } else {
    /*Si está la fila entonces solo hay que acomodar en la columna o reescribir el valor*/

    /*Buscamos el apuntador donde la columna sea menor al siguiente*/
    auto listaFilaIter = matrizIter->second.begin();
    while (listaFilaIter != matrizIter->second.end()) {
      if (listaFilaIter->first < columna)
        ++listaFilaIter;
      else
        break;
    }

    /*Si es igual entonces solo cambiar el valor*/

    if (listaFilaIter->first == columna) {
```

```
      listaFilaIter->second = valor;
      return;
    }

    /*Si no es igual entonces hay que insertar el valor*/
    std::pair<int, int> lista_fila_valor;
    lista_fila_valor.first = (int) columna;
    lista_fila_valor.second = valor;
    matrizIter->second.insert(listaFilaIter, lista_fila_valor);
  }
}
```

Implementación de suma de matrices

Para esta implementación haremos uso de la sobrecarga de operadores, para tener un operador + cuando sumemos entre dos clases `Matriz`.

Procedimiento:

1. Tenemos dos apuntadores al principio de la lista de fila de cada matriz, digamos `apuntadorFilas1` y `apuntadorFilas2`.
2. Si alguno de los apuntadores ya terminó, entonces agregar todos los restantes del apuntador que no acabó y terminar. Si ambos acabaron entonces terminar.
3. Si el valor de fila de alguno de los apuntadores está más adelantado que otro, entonces insertar el atrasado y aumentar su apuntador al siguiente elemento. Regresar a 2.
4. Si ambos apuntadores tienen el mismo valor de fila, entonces creamos los apuntadores de la lista de columna de ambas filas.
5. Igualmente que los de filas, si alguno de los apuntadores de columna ya terminó entonces agregar los elementos restantes de columna en la fila y terminar regresando a 8. Si ambos apuntadores de columna acabaron entonces terminar regresando a 8.
6. Si el valor de columna de alguno de los apuntadores está más adelantado que otro, entonces insertar el atrasado y aumentar su apuntador al siguiente elemento de columna. Regresar a 5.
7. Si ambos apuntadores tienen el mismo valor de columna, entonces agregamos en el resultado la suma de sus valores y aumentamos ambos apuntadores de la fila de columna a sus siguientes elementos.
8. Aumentar ambos apuntadores de fila y regresar a 2.

el código queda como sigue:

```
Matriz operator+(const Matriz &matriz1, const Matriz &matriz2) {

  /*Si las dimensiones no concuerdan entonces es un error*/
  if (!(matriz1._fila_dim == matriz2._fila_dim && matriz1._columna_dim == matriz2._columna_dim)) {
    throw std::runtime_error("No se pueden sumar matrices con dimensiones diferentes");
  }

  Matriz resultado(matriz1._fila_dim, matriz1._columna_dim);

  /*Suma*/

  /*Iteradores de las listas de filas*/
  auto matrix1It = matriz1._matriz.begin();
  auto matrix2It = matriz2._matriz.begin();

  while (true) {

    /* Si ya llegamos al final de la matriz 1 y faltan elementos en la matriz 2
     * entonces copiar los elementos restantes de la matriz 2
     * */
    if (matrix1It == matriz1._matriz.end() && matrix2It != matriz2._matriz.end()) {
      while (matrix2It != matriz2._matriz.end()) {
        resultado._matriz.push_back(*matrix2It++);
      }
      break;
      /* Si ya llegamos al final de la matriz 2 y faltan elementos en la matriz 1
       * entonces copiar los elementos restantes de la matriz 1
       * */
    } else if (matrix1It != matriz1._matriz.end() && matrix2It == matriz2._matriz.end()) {
      while (matrix1It != matriz1._matriz.end()) {
        resultado._matriz.push_back(*matrix1It++);
      }
      break;
      /* Si ya llegamos al final de ambos entonces damos por terminado todo
       * */
    } else if (matrix1It == matriz1._matriz.end() && matrix2It == matriz2._matriz.end()) {
```

```
      break;
      /*Si ninguno a llegado al final entonces ir avanzando*/
    } else {
      /*Alcanzar a la fila con mayor valor*/
      if (matrix1It->first < matrix2It->first) {
        resultado._matriz.push_back(*matrix1It++);
        continue;
      } else if (matrix1It->first > matrix2It->first) {
        resultado._matriz.push_back(*matrix2It++);
        continue;
      } else {
        /*Suma en lista de filas*/
        auto fila1 = matrix1It->second.begin();
        auto fila2 = matrix2It->second.begin();

        std::list<std::pair<unsigned int, int>> filaresult;
        while (true) {
          /*Si una fila acabó entonces hay que copiar las demás restantes*/
          if (fila1 == matrix1It->second.end() && fila2 != matrix2It->second.end()) {
            while (fila2 != matrix2It->second.end()) {
              filaresult.push_back(*fila2++);
            }
            break;
          } else if (fila2 == matrix2It->second.end() && fila1 != matrix1It->second.end()) {
            while (fila1 != matrix1It->second.end()) {
              filaresult.push_back(*fila1++);
            }
            break;
          } else if (fila2 == matrix2It->second.end() && fila1 == matrix1It->second.end()) {
            break;
          } else {
            if (fila1->first < fila2->first) {
              filaresult.push_back(*fila1++);
              continue;
            } else if (fila1->first > fila2->first) {
              filaresult.push_back(*fila2++);
              continue;
            } else {
              int sum = fila1->second + fila2->second;
              filaresult.emplace_back(fila1->first, sum);
              ++fila1, ++fila2;
            }
          }
        }

        resultado._matriz.emplace_back(matrix1It->first, filaresult);

        ++matrix1It, ++matrix2It;
      }
    }
  }
  return resultado;
}
```

Implementación de búsqueda de un valor

Esta función tiene el prototipo `int Matriz::GetValue(unsigned int fila, unsigned int columna)`, nos devuelve el valor que hay en esa fila y columna, si no lo encuentra devuelve 0.

Procedimiento:

1. Si la fila no existe, entonces regresa 0.
2. Si existe, entonces dentro de esa fila se busca la columna.
3. Si no existe la columna, entonces regresa 0.
4. Si existe la columna entonces solo regresamos el valor.

El código queda como sigue:

```
int Matriz::GetValue(unsigned int fila, unsigned int columna) const {
  int result = 0;
  /*Si los valores no están en donde deben estar entonces es un error de uso de la función*/
  if (!(0 <= fila < this->_fila_dim && 0 <= columna < this->_columna_dim))
```

```
  throw std::runtime_error("no existe tal fila o columna");

/*Buscar fila*/
auto matrizIter = std::find_if(this->_matriz.begin(), this->_matriz.end(),
 [&fila](const std::pair<unsigned int, std::list<std::pair<unsigned int, int>>> &el) {
   if (el.first == fila) {
     return true;
   } else {
     return false;
   }
 });

if (matrizIter != this->_matriz.end()) {
  /*Buscar columna si se encontró la fila*/
  auto ColumnaIter = std::find_if(matrizIter->second.begin(), matrizIter->second.end(),
  [&columna](const std::pair<unsigned int, int> &el) {
   if (el.first == columna) {
     return true;
   } else {
     return false;
   }
  });
  /*Regresar valor si se encontró la fila*/
  if(ColumnaIter != matrizIter->second.end()){
    result = ColumnaIter->second;
    }
  }

  return result;
}
```

Implementación de la multiplicación entre matrices y multiplicación de matriz por un escalar.

Igualmente usaremos sobrecarga de operadores.

Usamos la función privada anterior, para usar la multiplicación de matriz usual como sigue:

```
Matriz operator*(const Matriz &matriz1, const Matriz &matriz2) {
  /*Si las dimensiones no concuerdan entonces es un error*/
  if (matriz1._columna_dim != matriz2._fila_dim) {
    throw std::runtime_error("No se pueden sumar matrices con dimensiones diferentes");
  }

  Matriz resultado(matriz1._fila_dim, matriz2._columna_dim);

  /*Multiplicación*/
  for (int rowIndex = 0; rowIndex < resultado._fila_dim; ++rowIndex) {
    for (int columnIndex = 0; columnIndex < resultado._columna_dim; ++columnIndex) {
      int sum = 0;
      for (int rowcolumnIndex = 0; rowcolumnIndex < matriz1._columna_dim; ++rowcolumnIndex) {
        int a = matriz1.GetValue(rowIndex, rowcolumnIndex);
        int b = matriz2.GetValue(rowcolumnIndex, columnIndex);
        sum += a*b;
      }
      if(sum != 0){
        resultado.InsertElement(rowIndex, columnIndex, sum);
      }
    }
  }

  return resultado;
}
```

Y la multiplicación escalar es simplemente iterar toda la estructura multiplicando todos los valores en la lista de columnas por el escalar como sigue:

```
Matriz operator*(const int &value, const Matriz &matriz) {

  Matriz resultado(matriz);

  /*Multiplicación escalar*/
  for (auto &resFila: resultado._matriz) {
    for (auto &resColuma: resFila.second) {
```

```
      resColuma.second *= value;
    }
  }

  return resultado;
}
```

Resultados

Para los resultados usaremos las siguientes matrices, extraidas del archivo con los datos.

```
Matriz 2
0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 3, 0, 0,
0, 0, 0, 0, 1, 3, 6,
7, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
6, 0, 0, 0, 0, 0, 0,
0, 3, 0, 0, 8, 6, 0,

Matriz 1
1, 0, 7,
0, 0, 0,
0, 4, 0,
4, 0, 0,

Matriz 0
0, 5, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
0, 6, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 4,
0, 3, 0, 0, 0, 0, 0,
```

Suma

Sumamos la matriz 0 y 2, y el resultado obtenido lo sumamos consigo mismo. Obtenemos lo siguiente:

```
resultado de suma de la matriz 0 con la 2
0, 5, 0, 0, 0, 0, 0,
0, 0, 0, 1, 3, 0, 0,
0, 6, 0, 0, 1, 3, 6,
7, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
6, 0, 0, 0, 0, 0, 4,
0, 6, 0, 0, 8, 6, 0,

resultado de suma consigo mismo
0, 10, 0, 0, 0, 0, 0,
0, 0, 0, 2, 6, 0, 0,
0, 12, 0, 0, 2, 6, 12,
14, 0, 2, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
12, 0, 0, 0, 0, 0, 8,
0, 12, 0, 0, 16, 12, 0,
```

Multiplicación

Multiplicamos la matriz 2 con la 0, y el resultado obtenido lo multiplicamos consigo mismo. Obtenemos lo siguiente:

```
resultado de multiplicación de la matriz 2 con la 0
0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0,
0, 18, 0, 0, 0, 0, 12,
0, 35, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
0, 30, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 24,

resultado de multiplicación consigo mismo
```

```
0, 0, 0, 20, 60, 0, 0,
28, 0, 4, 0, 0, 0, 0,
72, 0, 0, 24, 264, 144, 48,
0, 164, 0, 0, 4, 12, 24,
0, 0, 0, 0, 0, 0, 0,
0, 216, 0, 0, 128, 96, 0,
144, 0, 0, 24, 72, 0, 96,
```

El cual es correcto corroborando con cualquier herramienta como python o wolfram alpha. Ahora bien por un escalar es el más sencillo de observar que está correcto.

Multiplicamos la matriz 2 por 5.

```
resultado de multiplicación de la matriz 2 con el número 5
0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 5, 15, 0, 0,
0, 0, 0, 0, 5, 15, 30,
35, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
30, 0, 0, 0, 0, 0, 0,
0, 15, 0, 0, 40, 30, 0,
```

Conclusiones

Los resultados muestran el buen comportamiento de las operaciones soportadas y requeridas en el proyecto, sin embargo, a falta de tiempo se puede haber optimizado el código de multiplicación con simplificaciones matemáticas. Aún así, el código tiene alta confiabilidad en este estado.