TDA Insieme

Algoritmi e Strutture Dati

Economia e Management

Prof. Carlo Blundo
DISA-MIS

Il TDA Insieme

- Vogliamo implementare un TDA che conserva degli elementi come un insieme
- In Python esiste il TDA set, implementeremo alcune delle funzionalità di set
- Usiamo, come rappresentazione interna, una lista

Operazioni

- costruttore
- len
- in
- inserisci e cancella
- sottoinsiemeDi
- == e !=
- unione e intersezione
- iteratore

Metodi in Python

```
class Insieme:
    def __init__(self):
        pass

    def __len__(self):
        pass

    def __contains__(self, element):
        pass

    def inserisci(self, element):
        pass

    def cancella(self, element):
        pass

    def __eq__(self, setB):
        pass

    def __ne__(self, setB):
        pass

    def sottoinsiemeDi(self, setB):
        pass

    def unione(self, setB):
        pass

    def intersezione(self, setB):
        pass

    def __iter__(self):
        pass

    def __next__(self):
        pass
```

Implementazione

Complessità

```
class Insieme:
    def __init__(self):
        self._theElements = list()

    def __len__(self):
        return len(self._theElements)

    def __contains__(self, elemento):
        return elemento in self._theElements

    def __str__(self):
        return str(self._theElements)
```

O(1)

O(1)

O(n)

O(n)

La complessità di tempo dei metodi dipende dalla complessità di tempo dei metodi di list

Implementazione

Complessità

```
def inserisci(self, elemento):
    if elemento not in self:
        self._theElements.append(elemento)
```
O(n)

```
def cancella(self, element):
    assert element in self, "L'elemento non appartiene all'insieme"
    self._theElements.remove(element)
```
O(n)

```
def __eq__(self, setB):
    if len(self) != len(setB):
        return False
    else:
        return self.sottoinsiemeDi(setB)
```
$O(n^2)$

```
def __ne__(self, setB):
    return not self == setB
```
$O(n^2)$

```
def sottoinsiemeDi(self, setB):
    for elemento in self:
        if elemento not in setB:
            return False
    return True
```
$O(n^2)$

```
def unione(self, setB):
    nuovo_insieme = Insieme()
    for elemento in self:
        nuovo_insieme.inserisci(elemento)
    for elemento in setB:
        if elemento not in nuovo_insieme:
            nuovo_insieme.inserisci(elemento)
    return nuovo_insieme

def intersezione(self, setB):
    nuovo_insieme = Insieme()
    for elemento in self:
        if elemento in setB:
            nuovo_insieme.inserisci(elemento)
    return nuovo_insieme


# iterator
def __iter__(self):
    self._pos = 0
    return self

# iterator
def __next__(self):
    if self._pos < len(self._theElements):
        item = self._theElements[self._pos]
        self._pos += 1
        return item
    else:
        raise StopIteration
```

Complessità

$O(n^2)$

$O(n^2)$

$O(1)$

$O(n)$

Matrice Sparsa

Matrice Sparsa

- Una matrice **sparsa** è una matrice che contiene tanti elementi uguali a 0.
- Una matrice n x m è detta **sparsa** se contiene solo k (con k << n x m) elementi diversi da 0
- Le matrici sparse sono molto comuni in applicazioni scientifiche, specialmente quelle che utilizzano sistemi di equazioni lineari
- Una rappresentazione *tradizionale* comporta uno spreco di memoria

Esempio

$$\begin{bmatrix} \cdot & 3 & \cdot & \cdot & 8 & \cdot & \cdot & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & \cdot & 5 & \cdot \\ \cdot & \cdot & 9 & \cdot & \cdot & 2 & \cdot & \cdot \\ \cdot & 7 & \cdot & \cdot & \cdot & \cdot & \cdot & 3 \\ \cdot & \cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot \end{bmatrix}$$

Matrice 5x8 con 10 elementi diversi da zero

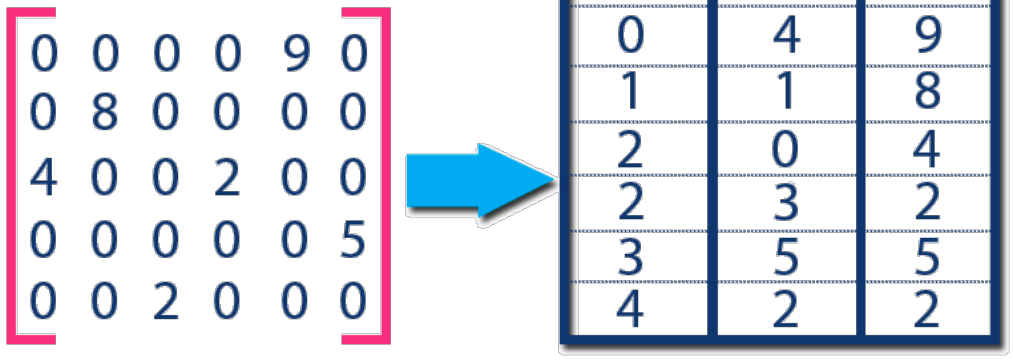
I punti rappresentano gli elementi uguali a zero

Con la rappresentazione tradizionale si occupa uno spazio proporzionale a 40, mentre sarebbe necessario memorizzare solo 10 elementi

Rappresentazione

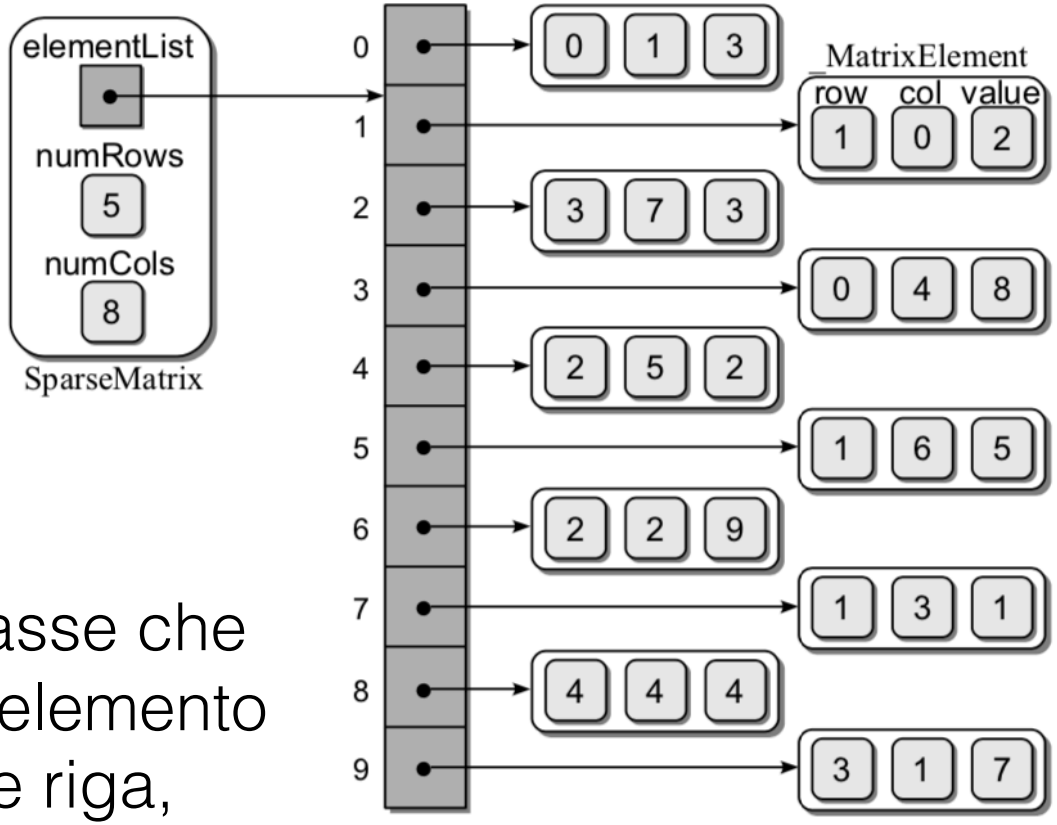
- Possiamo cambiare rappresentazione
- Utilizziamo delle triple per rappresentare gli elementi della matrice
  - (riga, colonna, elemento)

La prima tripla nella tabella rappresenta il numero di righe, di colonne e di elementi diversi da zero



Rappresentazione

Le triple le possiamo memorizzare in una lista



_MatrixElement è una classe che rappresenta un generico elemento della matrice. Contiene riga, colonna ed elemento

_MatrixElement e SparseMatrix

Per semplicità non definiamo setter e getter della classe _MatrixElement

```
class SparseMatrix:
    class _MatrixElement:
        def __init__(self, row, col, value):
            self._row = row
            self._col = col
            self._value = value

    # Crea una matrice numRows x numCols sparsa inizializzata a 0
    def __init__(self, numRows, numCols):
        self._numRows = numRows
        self._numCols = numCols
        self._elementList = list()

    # Restituisce il numero di righe della matrice
    def numRows( self ): return self._numRows

    # Restituisce il numero di colonne della matrice
    def numCols( self ): return self._numCols
```

_findPosition

```
def _findPosition(self, row, col):
    i = 0
    for element in self._elementList:
        if row == element._row and col == element._col:
            return i
        i += 1
    return None
```

Metodo ausiliario utilizzato per cercare la posizione dell'elemento in riga row e colonna col

Il metodo restituisce None se tale elemento non si trova (nella matrice originale è uguale a zero)

__setitem__ e __getitem__

- Data una matrice m di tipo SparseMatrix
- __setitem__
  - Vogliamo poter assegnare un valore in posizione (i,j) nel seguente modo m[i, j] = 9
- __setitem__
  - Analogamente, vogliamo poter accedere al valore in posizione (i,j) nel seguente modo a = m[i, j]

```
def __setitem__(self, key, value):
    print(key, value)
```

__setitem__

```
m[4] = 1          →  4 1
m[2, 8] = 5       →  (2, 8) 5
m[3, 5, 4] = 7    →  (3, 5, 4) 7
```

- Ha tre parametri: self, key e value
  - self: sappiamo cosa rappresenta
  - key: rappresenta la posizione del contenitore dove inserire l'elemento a destra dell'uguale. Se tra le parentesi quadre inseriamo un solo elemento, key assume il valore dell'elemento, altrimenti key è una tupla che contiene tutti gli elementi passati
  - value: il valore da inserire nel contenitore in posizione key

__setitem__

```
def __setitem__(self, ndxTuple, scalar):
    ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
    if ndx is not None:  # se l'elemento è trovato in _elementList
        # se il nuovo valore è diverso da zero, modifichiamo quello vecchio
        if scalar != 0.0:
            self._elementList[ndx]._value
        else:                    #altrimenti rimuoviamo l'elemento da _elementList
            self._elementList.pop(ndx)
    else:  # se l'elemento non nella lista ed è diverso da zero
        if scalar != 0.0 :
            # aggiungiamo l'elemento a _elementList
            element = self._MatrixElement(ndxTuple[0], ndxTuple[1], scalar)
            self._elementList.append(element)
```

Se _MatrixElement è definito al di fuori della classe SparseMatrix, non è necessario qualificarlo con self

__getitem__

```
def __getitem__(self, ndxTuple):
    print(ndxTuple)
```

```
m[4]          →  4
m[3, 5]       →  (3, 5)
m[3, 5, 3]    →  (3, 5, 3)
```

- __getitem__ verifica c'è un valore in posizione (i,j) invocando il metodo _findPosition. Se _findPosition restituisce None, __getitem__ restituisce 0, altrimenti restituisce il valore associato a (i,j)

```
# Restituisce il valore dell'elemento (i, j), x[i, j]
def __getitem__(self, ndxTuple):
    ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
    return self._elementList[ndx]._value if ndx != None else 0
```

Metodo __add__

A + B

- Verificare che le due matrici A ed B hanno dimensioni compatibili (stesso numero di righe e stesso numero di colonne)
- Creare una nuova istanza N di SparseMatrix con lo stesso numero di righe e e colonne di A
- Duplicare gli elementi della matrice self (A) e copiarli nella matrice N (non si possono copiarli direttamente, altrimenti si copia solo il loro riferimento)
- Iterare sugli elementi di B e addizionarli ai corrispondenti elementi della matrice N

Metodo __add__

```
def __add__(self, B):
    assert B.numRows() == self.numRows() and \
           B.numCols() == self.numCols(), \
           "Le matrici non sono compatibili per essere addizionate."
    # Creare la nuova matrice.
    N = SparseMatrix(self.numRows(), self.numCols())

    # Duplicare la matrice self, _MatrixElement è mutable, non
    # possiamo semplicemente copiare il riferimento
    for elemento in self._elementList :
        dupElement = self._MatrixElement(elemento._row, elemento._col, elemento._value)
        N._elementList.append( dupElement )

    # Iterare sugli elementi diversi da zero della matrice B
    for elemento in B._elementList :
        # Prendere il corrispondente valore
        valore = N[elemento._row, elemento._col]
        valore += elemento._value
        # Memorizzare il nuovo valore nella nuova matrice
        N[elemento._row, elemento._col ] = valore

    # Restituire la nuova matrice
    return N
```

se aggiungiamo l'iteratore, possiamo scrivere `for elemento in B`

Metodi __sub__ e __mul__

- Implementazione analoga a quella del metodo __add__
- Ci sono delle differenze per il metodo __mul__
  - Occorrono tre cicli di for annidati
  - Le matrici sono compatibili quando
    - A è n x k e
    - B è k x n
- Implementare __sub__ e __mul__

Complessità

Dipende dalla complessità dei metodi di list e dal modo in cui sono utilizzati

```
class SparseMatrix:
    class _MatrixElement:
        def __init__(self, row, col, value):
            self._row = row
            self._col = col
            self._value = value

    # Crea una matrice numRows x numCols sparsa inizializzata a 0
    def __init__(self, numRows, numCols):
        self._numRows = numRows
        self._numCols = numCols
        self._elementList = list()

    # Restituisce il numero di righe della matrice
    def numRows( self ): return self._numRows

    # Restituisce il numero di colonne della matrice
    def numCols( self ): return self._numCols
```

O(1)

O(1)

O(1)

O(1)

Complessità

k: numero di elementi della matrice diversi da zero

```
def _findPosition(self, row, col):
    i = 0
    for element in self._elementList:
        if row == element._row and col == element._col:
            return i
        i += 1
    return None
```

O(k)

```
  # Restituisce il valore dell'elemento (i, j), x[i, j]
  def __getitem__(self, ndxTuple):
      ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
      return self._elementList[ndx]._value if ndx != None else 0
```

O(k)

Complessità

k: numero di elementi della matrice diversi da zero

O(k)

```
def __setitem__(self, ndxTuple, scalar):
    ndx = self._findPosition(ndxTuple[0], ndxTuple[1])
    if ndx is not None:  # se l'elemento è trovato in _elementList
        # se il nuovo valore è diverso da zero, modifichiamo quello vecchio
        if scalar != 0.0:
            self._elementList[ndx]._value = = scalar
        else:                    #altrimenti rimuoviamo l'elemento da _elementList
            self._elementList.pop(ndx)
    else:  # se l'elemento non nella lista ed è diverso da zero
        if scalar != 0.0 :
            # aggiungiamo l'elemento a _elementList
            element = self._MatrixElement(ndxTuple[0], ndxTuple[1], scalar)
            self._elementList.append(element)
```

O(1)

O(ndx)

O(k)

$$O(k) + O(1) + O(ndx) + O(k) = O(k)$$

k: numero di elementi non nulli della matrice self

k´: numero di elementi non nulli della matrice B

Assumiamo
O(k) = O(k´)

Complessità

```
    def __add__(self, B):
O(1)    assert B.numRows() == self.numRows() and \
               B.numCols() == self.numCols(), \
               "Le matrici non sono compatibili per essere addizionate."
        # Creare la nuova matrice.
O(1)    N = SparseMatrix(self.numRows(), self.numCols())

        # Duplicare la matrice self, _MatrixElement è mutable, non
        # possiamo semplicemente copiare il riferimento
O(k)    for elemento in self._elementList :
            dupElement = self._MatrixElement(elemento._row, elemento._col, elemento._value)
  O(1)      N._elementList.append( dupElement )

        # Iterare sugli elementi diversi da zero della matrice B
O(k´)   for elemento in B._elementList :
            # Prendere il corrispondente valore
  O(k)      valore = N[elemento._row, elemento._col]
            valore += elemento._value
            # Memorizzare il nuovo valore nella nuova matrice
  O(k)      N[elemento._row, elemento._col ] = valore

        # Restituire la nuova matrice
        return N
```

$$O(1) + O(1) + O(k) + O(k´)*[O(k) + O(k)] = O(k^2)$$