Sparse Matrices and Their Data Structures
(PSC §4.2)
Basic sparse technique: adding two vectors

- Problem: add a sparse vector $\mathbf{y}$ of length $n$ to a sparse vector $\mathbf{x}$ of length $n$, overwriting $\mathbf{x}$, i.e.,

$$\mathbf{x} := \mathbf{x} + \mathbf{y}.$$ 

- $\mathbf{x}$ is a **sparse vector** means that $x_i = 0$ for most $i$.

- The number of nonzeros of $\mathbf{x}$ is $c_x$ and that of $\mathbf{y}$ is $c_y$. 
Example: storage as compressed vector

- Vectors \( \mathbf{x} \), \( \mathbf{y} \) have length \( n = 8 \).
- Their number of nonzeros is \( c_\mathbf{x} = 3 \) and \( c_\mathbf{y} = 4 \).
- A **compressed vector** data structure for \( \mathbf{x} \) and \( \mathbf{y} \) is:

  \[
  \begin{array}{|c|c|c|c|}
  \hline
  x[j].a & 2 & 5 & 1 \\
  x[j].i & 5 & 3 & 7 \\
  \hline
  y[j].a & 1 & 4 & 1 & 4 \\
  y[j].i & 6 & 3 & 5 & 2 \\
  \hline
  \end{array}
  \]

- Here, the \( j \)th nonzero in the array of \( \mathbf{x} \) has
  numerical value \( x_i = x[j].a \) and index \( i = x[j].i \).
- How to compute \( \mathbf{x} + \mathbf{y} \)?
Addition is easy for dense storage

- The **dense vector** data structure for \( x, y, \) and \( x + y \) is:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- A compressed vector data structure for \( z = x + y \) is:

\[
\begin{align*}
z[j].a &= 3 & 9 & 1 & 1 & 4 \\
z[j].i &= 5 & 3 & 7 & 6 & 2
\end{align*}
\]

- Conclusion: use an **auxiliary dense vector**!
Location array

The array \( \text{loc} \) (initialised to \(-1\)) stores the location \( j = \text{loc}[i] \) where a nonzero vector component \( y_i \) is stored in the compressed array.

\[
\begin{array}{c|cccc}
 y[j].a & 1 & 4 & 1 & 4 \\
 y[j].i & 6 & 3 & 5 & 2 \\
 j & 0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c|ccccccccc}
 y_i & 0 & 0 & 4 & 4 & 0 & 1 & 1 & 0 \\
 \text{loc}[i] & -1 & -1 & 3 & 1 & -1 & 2 & 0 & -1 \\
 i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
Algorithm for sparse vector addition: pass 1

**input:**
- \( \mathbf{x} \): sparse vector with \( c_x \) nonzeros, \( \mathbf{x} = \mathbf{x}_0 \),
- \( \mathbf{y} \): sparse vector with \( c_y \) nonzeros,
- \( \mathbf{loc} \): dense vector of length \( n \),
  \( \mathbf{loc}[i] = -1 \), for \( 0 \leq i < n \).

**output:**
- \( \mathbf{x} = \mathbf{x}_0 + \mathbf{y} \),
- \( \mathbf{loc}[i] = -1 \), for \( 0 \leq i < n \).

\[
\{ \text{Register location of nonzeros of } \mathbf{y} \} \\
\text{for } j := 0 \text{ to } c_y - 1 \text{ do} \\
\quad \mathbf{loc}[\mathbf{y}[j].i] := j;
\]
Algorithm for sparse vector addition: passes 2, 3

\{ Add matching nonzeros of \( x \) and \( y \) into \( x \}\}

\begin{verbatim}
for \( j := 0 \) to \( c_x - 1 \) do
    \( i := x[j].i; \)
    if \( \text{loc}[i] \neq -1 \) then
        \( x[j].a := x[j].a + y[\text{loc}[i]].a; \)
        \( \text{loc}[i] := -1; \)
\end{verbatim}
Algorithm for sparse vector addition: passes 2, 3

{ Add matching nonzeros of \( \mathbf{x} \) and \( \mathbf{y} \) into \( \mathbf{x} \) }

\[
\text{for } j := 0 \text{ to } c_\mathbf{x} - 1 \text{ do } \\
i := x[j].i; \\
\text{if } loc[i] \neq -1 \text{ then } \\
x[j].a := x[j].a + y[loc[i]].a; \\
loc[i] := -1;
\]

{ Append remaining nonzeros of \( \mathbf{y} \) to \( \mathbf{x} \) }

\[
\text{for } j := 0 \text{ to } c_\mathbf{y} - 1 \text{ do } \\
i := y[j].i; \\
\text{if } loc[i] \neq -1 \text{ then } \\
x[c_\mathbf{x}].i := i; \\
x[c_\mathbf{x}].a := y[j].a; \\
c_\mathbf{x} := c_\mathbf{x} + 1; \\
loc[i] := -1;
\]
Analysis of sparse vector addition

- The total number of operations is $O(c_x + c_y)$, since there are $c_x + 2c_y$ loop iterations, each with a small constant number of operations.

- The number of flops equals the number of nonzeros in the intersection of the sparsity patterns of $x$ and $y$. 0 flops can happen!

- Initialisation of array $loc$ costs $n$ operations, which will dominate the total cost if only one vector addition has to be performed.

- $loc$ can be reused in subsequent vector additions, because each modified element $loc[i]$ is reset to $-1$.

- If we add two $n \times n$ matrices row by row, we can amortise the $O(n)$ initialisation cost over $n$ vector additions.
Accidental zero

17,758 $\times$ 17,758 matrix memplus with 126,150 entries, including 27,003 accidental zeros.

- An accidental zero is a matrix element that is numerically zero but still occurs as a nonzero pair $(i, 0)$ in the data structure.
- Accidental zeros are created when a nonzero $y_i = -x_i$ is added to a nonzero $x_i$ and the resulting zero is retained.
- Testing all operations in a sparse matrix algorithm for zero results is more expensive than computing with a few additional nonzeros.
- Therefore, accidental zeros are usually kept.
No abuse of numerics for symbolic purposes!

- Instead of using the symbolic location array, initialised at $-1$, we could have used an auxiliary array storing numerical values, initialised at 0.0.
- We could then add $y$ into the numerical array, update $x$ accordingly, and reset the array.
- Unfortunately, this would make the resulting sparsity pattern of $x + y$ dependent on the numerical values of $x$ and $y$: an accidental zero in $y$ would never lead to a new entry in the data structure of $x + y$.
- This dependence may prevent reuse of the sparsity pattern in case the same program is executed repeatedly for a matrix with different numerical values but the same sparsity pattern.
- Reuse often speeds up subsequent program runs.
Sparse matrix data structure: coordinate scheme

- In the **coordinate scheme** or **triple scheme**, every nonzero element \( a_{ij} \) is represented by a triple \((i, j, a_{ij})\), where \( i \) is the row index, \( j \) the column index, and \( a_{ij} \) the numerical value.
- The triples are stored in arbitrary order in an array.
- This data structure is easiest to understand and is often used for **input/output**.
- It is suitable for **input to a parallel computer**, since all information about a nonzero is contained in its triple. The triples can be sent directly and independently to the responsible processors.
- Row-wise or column-wise operations on this data structure require a lot of searching.
Compressed Row Storage

- In the **Compressed Row Storage** (CRS) data structure, each matrix row \( i \) is stored as a compressed sparse vector consisting of pairs \((j, a_{ij})\) representing nonzeros.
- In the data structure, \( a[k] \) denotes the numerical value of the \( k \)th nonzero, and \( j[k] \) its column index.
- Rows are stored consecutively, in order of increasing \( i \).
- \( start[i] \) is the address of the first nonzero of row \( i \).
- The number of nonzeros of row \( i \) is \( start[i + 1] - start[i] \), where by convention \( start[n] = nz(A) \).
Example of CRS

\[ A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 8 & 9 \end{bmatrix}, \quad n = 5, \quad nz(A) = 13. \]

The CRS data structure for \( A \) is:

- \( a[k] = [3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ 5 \ 3 \ 5 \ 8 \ 9] \)
- \( j[k] = [1 \ 4 \ 0 \ 1 \ 1 \ 2 \ 3 \ 0 \ 3 \ 4 \ 2 \ 3 \ 4] \)
- \( k = [0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12] \)
- \( \text{start}[i] = [0 \ 2 \ 4 \ 7 \ 10 \ 13] \)
- \( i = [0 \ 1 \ 2 \ 3 \ 4 \ 5] \)
Sparse matrix–vector multiplication using CRS

**input:** $A$: sparse $n \times n$ matrix,
$v$: dense vector of length $n$.

**output:** $u$: dense vector of length $n$, $u = Av$.

\[
\text{for } i := 0 \text{ to } n - 1 \text{ do}
\]
\[
\quad u[i] := 0;
\]
\[
\quad \text{for } k := \text{start}[i] \text{ to } \text{start}[i + 1] - 1 \text{ do}
\]
\[
\quad \quad u[i] := u[i] + a[k] \cdot v[j[k]];
\]
Incremental Compressed Row Storage (ICRS) is a variant of CRS proposed by Joris Koster in 2002.

In ICRS, the location \((i, j)\) of a nonzero \(a_{ij}\) is encoded as a 1D index \(i \cdot n + j\).

Instead of the 1D index itself, the difference with the 1D index of the previous nonzero is stored, as an increment in the array \(inc\). This technique is sometimes called delta indexing.

The nonzeros within a row are ordered by increasing \(j\), so that the 1D indices form a monotonically increasing sequence and the increments are positive.

An extra dummy element \((n, 0)\) is added at the end.
Example of ICRS

\[
A = \begin{bmatrix}
0 & 3 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 \\
0 & 5 & 9 & 2 & 0 \\
6 & 0 & 0 & 5 & 3 \\
0 & 0 & 5 & 8 & 9 \\
\end{bmatrix}, \quad n = 5, \quad nz(A) = 13.
\]

The ICRS data structure for \( A \) is:

| \( a[k] \) | 3 | 1 | 4 | 1 | 5 | 9 | 2 | ... | 0 |
| \( j[k] \) | 1 | 4 | 0 | 1 | 1 | 2 | 3 | ... | 0 |
| \( i[k] \cdot n + j[k] \) | 1 | 4 | 5 | 6 | 11 | 12 | 13 | ... | 25 |
| \( inc[k] \) | 1 | 3 | 1 | 1 | 5 | 1 | 1 | ... | 1 |
| \( k \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | 13 |
Sparse matrix–vector multiplication using ICRS

\[ \text{input: } \quad A: \text{ sparse } n \times n \text{ matrix,} \]
\[ \quad v: \text{ dense vector of length } n. \]
\[ \text{output: } \quad u: \text{ dense vector of length } n, \quad u = Av. \]

\[ k := 0; \quad j := inc[0]; \]
\[ \text{for } i := 0 \text{ to } n - 1 \text{ do} \]
\[ \quad u[i] := 0; \]
\[ \quad \text{while } j < n \text{ do} \]
\[ \quad \quad u[i] := u[i] + a[k] \cdot v[j]; \]
\[ \quad \quad k := k + 1; \]
\[ \quad \quad j := j + inc[k]; \]
\[ \quad \quad j := j - n; \]

Slightly faster: increments translate well into pointer arithmetic of programming language C; no indirect addressing \( v[j[k]] \).
A few other data structures

- **Compressed column storage (CCS)**, similar to CRS.
- **Gustavson’s data structure**: both CRS and CCS, but storing numerical values only once. Offers row-wise and column-wise access to the sparse matrix.
- The two-dimensional doubly linked list: each nonzero is represented by \(i, j, a_{ij}\), and links to a next and a previous nonzero in the same row and column. Offers maximum flexibility: row-wise and column-wise access are easy and elements can be inserted and deleted in \(O(1)\) operations.
- **Matrix-free storage**: sometimes it may be too costly to store the matrix explicitly. Instead, each matrix element is recomputed when needed. Enables solution of huge problems.
Summary

- Sparse matrix algorithms are **more complicated** than their dense equivalents, as we saw for sparse vector addition.
- Sparse matrix computations have a **larger integer overhead** associated with each floating-point operation.
- Still, using sparsity can save large amounts of CPU time and also memory space.
- We learned an efficient way of adding two sparse vectors using a dense initialised auxiliary array. You will be surprised to see how often you can use this trick.
- **Compressed row storage** (CRS) and its variants are useful data structures for sparse matrices.
- CRS stores the nonzeros of each row together, but does not sort the nonzeros within a row. Sorting is a mixed blessing: it may help, but it also takes time.