Graphs
How could this (directed) graph be written as a matrix?

\[
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1
\end{pmatrix}
\]
What is the general rule for turning a graph into a matrix?

If there is an edge from node $i$ to node $j$, then $A_{ji} = 1$.
(otherwise zero)

What does the matrix for an undirected graph look like?

Symmetric.
How could we turn a *weighted graph* (i.e. one where the edges have weights—maybe ‘pipe widths’) into a matrix?

Allow values other than zero and one for the entries of the matrix.
If we multiply a graph matrix by the $i$th unit vector, what happens?
We get a vector that indicates (with a 1) all the nodes that are reachable from node $i$. 

\[
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{pmatrix}
\begin{pmatrix}
1 \\
0 \\
0 \\
0 \\
0 \\
\end{pmatrix}
= 
\begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
\end{pmatrix}.
\]
Consider the following graph of states:

Suppose this is an accurate model of the behavior of the average student. :) Can this be described using a matrix?
Important assumption: Only the most recent state matters to determine probability of next state. This is called the Markov property, and the model is called a Markov chain.

Write transition probabilities into matrix as before: (Order: surf, study, eat—‘from’ state along columns)

\[
A = \begin{pmatrix}
0.8 & 0.6 & 0.8 \\
0.2 & 0.3 & 0 \\
0 & 0.1 & 0.2 \\
\end{pmatrix}
\]

Observe: Columns add up to 1, to give sensible probability distribution of next states. Given probabilities of states \( p = (p_{\text{surf}}, p_{\text{study}}, p_{\text{eat}}) \), \( Ap \) gives us the probabilities after one unit of time has passed.
PageRank

Example
Problem: Consider $n$ linked webpages. Rank them.

- Let $x_1, \ldots, x_n \geq 0$ represent importance
- A link to a page increases the perceived importance of a webpage

Example
Try $n = 4$.

- page 1: 2,3,4
- page 2: 3,4
- page 3: 1
- page 4: 1,3
First attempt

• Let $x_k$ be the number of links to page $k$
• Problem: a link from an important page like The NY Times has no more weight than lukeo.cs.illinois.edu
Second attempt

• Let $x_k$ be the sum of importance scores of all pages that link to page $k$
• Problem: a webpage has more influence simply by having more outgoing links
• Problem: the linear system is trivial (oops!)
Third attempt (Brin/Page ’90s)

• Let $n_j$ be the number of outgoing links on page $j$

• Let

$$x_k = \sum_{j \text{ linking to } k} \frac{x_j}{n_j}$$

• The influence of a page is its importance. It is split evenly to the pages it links to.

**Example**

Let $A$ be an $n \times n$ matrix as

$$A_{ij} = \begin{cases} 1/n_j & \text{if page } j \text{ links to page } i \\ 0 & \text{otherwise} \end{cases}$$
Page Rank

• Sum of column $j$ is $n_j/n_j = 1$, so $A$ is a Markov Matrix
• Problem: does not guarantee a unique $x$ s.t. $Ax = x$
• Brin-Page: Use instead

$$A \leftarrow 0.85A + 0.15$$

• Still a Markov Matrix
• Now has all positive entries
• Guarantees a unique solution
Page Rank

\[ A \leftarrow 0.85A + 0.15 \]

- What does this mean though?
- This defines a stochastic process: “PageRank can be thought of as a model of user behavior. We assume there is a random surfer who is given a web page at random and keeps clicking on links, never hitting back, but eventually gets bored and starts on another random page.”
- So a surfer clicks on a link on the current page with probability 0.85 and opens a random page with probability 0.15.
- PageRank is the probability that the random user will end up on that page.
Theory

Theorem
Perron-Frobenius If \( M \) is a Markov matrix with positive entries, then \( M \) has a unique steady-state vector \( x \).

Theorem
Perron-Frobenius Corollary Given an initial state \( x_0 \), then \( x_k = M^k x_0 \) converges to \( x \).
Markov Application

We consider a light weight version of computing a realistic BCS ranking. One difficult aspect of the BCS rankings for college football is that not every team plays each other.

- Consider a simpler version of ranking Big Ten teams after the first four weeks of play.
- Not every team has played each other (or even played another Big Ten Team).
Consider the following games:

<table>
<thead>
<tr>
<th>Michigan</th>
<th>16</th>
<th>Purdue</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iowa</td>
<td>38</td>
<td>Wisconsin</td>
<td>17</td>
</tr>
<tr>
<td>Iowa</td>
<td>28</td>
<td>Illinois</td>
<td>23</td>
</tr>
<tr>
<td>Minnesota</td>
<td>34</td>
<td>Michigan</td>
<td>21</td>
</tr>
<tr>
<td>Minnesota</td>
<td>23</td>
<td>Purdue</td>
<td>10</td>
</tr>
<tr>
<td>Purdue</td>
<td>31</td>
<td>Michigan</td>
<td>6</td>
</tr>
<tr>
<td>Wisconsin</td>
<td>33</td>
<td>Illinois</td>
<td>25</td>
</tr>
<tr>
<td>Wisconsin</td>
<td>38</td>
<td>Purdue</td>
<td>23</td>
</tr>
<tr>
<td>Illinois</td>
<td>27</td>
<td>Iowa</td>
<td>6</td>
</tr>
<tr>
<td>Illinois</td>
<td>20</td>
<td>Wisconsin</td>
<td>12</td>
</tr>
</tbody>
</table>
The adjacency matrix for this problem is:

\[
A_{i,j} = \begin{cases} 
 w_{i,j} & \text{team } i \text{ beats team } j \\
 0 & \text{otherwise}
\end{cases}
\]

where \( w_{i,j} \) is the absolute value of the difference between scores. Order the teams 1-Michigan, 2-Iowa, 3-Minnesota, 4-Purdue, 5-Wisconsin, 6-Illinois. Now, \( w_{1,3} \) represents a victory by Michigan over Minnesota by the amount assigned to \( w_{1,3} \).
As with the Google matrix we need the column sums to be one (to guarantee values of the eigenvector to be in \([0, 1]\)), so let

\[
H_{i,j} = \frac{1}{\sum_{k=1}^{n} A_{k,j}} A_{i,j}
\]

where we ignore any zero columns.
Markov Application

With this we have

\[
A = \begin{bmatrix}
0 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 21 & 5 \\
13 & 0 & 0 & 13 & 0 & 0 \\
25 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 15 & 0 & 8 \\
0 & 21 & 0 & 0 & 8 & 0 \\
\end{bmatrix}
\]

and

\[
H = \begin{bmatrix}
0 & 0 & 0 & 3/31 & 0 & 0 \\
0 & 0 & 0 & 0 & 21/29 & 5/13 \\
13/38 & 0 & 0 & 13/31 & 0 & 0 \\
25/38 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 15/31 & 0 & 8/13 \\
0 & 1 & 0 & 0 & 8/29 & 0 \\
\end{bmatrix}
\]
Markov Application

Teams that are undefeated have zero columns in $H$. We transform $H$ into a *stochastic matrix* (columns add to 1) by performing a rank-1 update:

$$H \leftarrow H + ua^T.$$ 

Letting $a$ be 1 for undefeated teams and 0 otherwise and $u$ be $1/6$ we have

$$a = [001000]^T \quad u = (1/6)[111111]^T$$

Thus

$$H + ua^T = \begin{bmatrix}
0 & 0 & 1/6 & 3/31 & 0 & 0 \\
0 & 0 & 1/6 & 0 & 21/29 & 5/13 \\
13/38 & 0 & 1/6 & 13/31 & 0 & 0 \\
25/38 & 0 & 1/6 & 0 & 0 & 0 \\
0 & 0 & 1/6 & 15/31 & 0 & 8/13 \\
0 & 1 & 1/6 & 0 & 8/29 & 0
\end{bmatrix}.$$
Markov Application

As with PageRank, we have a probability parameter $\alpha = 0.85$. In the BCS rankings, this would correspond to the likelihood that a voter would change their vote based on a loss to a higher ranked team. The final Google-like matrix is then

$$G = \alpha(H + ua^T) + (1 - \alpha)(1/6)ee^T$$

or

$$G = 0.85 \begin{bmatrix}
    0 & 0 & 1/6 & 3/31 & 0 & 0 \\
    0 & 0 & 1/6 & 0 & 21/29 & 5/13 \\
    13/38 & 0 & 1/6 & 13/31 & 0 & 0 \\
    25/38 & 0 & 1/6 & 0 & 0 & 0 \\
    0 & 0 & 1/6 & 15/31 & 0 & 8/13 \\
    0 & 1 & 1/6 & 0 & 8/29 & 0 \\
\end{bmatrix} + 0.15 \begin{bmatrix}
    1 & 1 & 1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}$$
Applying the power method to the matrix $G$ gives the team rankings. The number one team corresponds to the largest element of the eigenvector. After 20 iterations beginning with a random vector the normalized eigenvector is:

\[
\begin{bmatrix}
0.0780 \\
0.5663 \\
0.1315 \\
0.1124 \\
0.4590 \\
0.6577
\end{bmatrix} \rightarrow
\begin{bmatrix}
\text{Illinois} \\
\text{Iowa} \\
\text{Wisconsin} \\
\text{Minnesota} \\
\text{Purdue} \\
\text{Michigan}
\end{bmatrix}
\]

This shows that the team listed in position six has the largest component and it therefore first in the ranking.
Sparsity
Sparse Matrices

• Vague definition: matrix with few nonzero entries
• For all practical purposes: an $m \times n$ matrix is sparse if it has $O(\min (m, n))$ nonzero entries.
• This means roughly a constant number of nonzero entries per row and column
Sparse Matrices

• Other definitions use a slow growth of nonzero entries with respect to $n$ or $m$.

• Wilkinson’s Definition: “..matrices that allow special techniques to take advantage of the large number of zero elements.” (J. Wilkinson)

• Applications which lead to sparse matrices: Structural Engineering, Computational Fluid Dynamics, Reservoir simulation, Electrical Networks, optimization, data analysis, information retrieval (LSI), circuit simulation, device simulation, ...
Sparse Matrices: The Goal

- To perform standard matrix computations economically i.e., without storing the zeros of the matrix.
- For typical Finite Element /Finite difference matrices, number of nonzero elements is $O(n)$.

**Example**
To add two square dense matrices of size $n$ requires $O(n^2)$ operations. To add two sparse matrices $A$ and $B$ requires $O(nnz(A) + nnz(B))$ where $nnz(X)$ = number of nonzero elements of a matrix $X$.

**remark**
$A^{-1}$ is usually dense, but $L$ and $U$ in the $LU$ factorization may be reasonably sparse (if a good technique is used).
Sparse Matrices

- So how do we store $A$?
- Fast mat-vec is certainly important; also ask
  - what type of access (rows, cols, diag, etc)?
  - dynamic allocation?
  - transpose needed?
  - inherent structure?
- Even data structures for dense storage not as obvious
- Sparse operations have low operation/memory reference ratio
# Popular Storage Structures

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>Dense</td>
</tr>
<tr>
<td>BND</td>
<td>Linpack Banded</td>
</tr>
<tr>
<td>COO</td>
<td>Coordinate</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>CSC</td>
<td>Compressed Sparse Column</td>
</tr>
<tr>
<td>MSR</td>
<td>Modified CSR</td>
</tr>
<tr>
<td>LIL</td>
<td>Linked List</td>
</tr>
<tr>
<td>ELL</td>
<td>Ellpack-Itpack</td>
</tr>
<tr>
<td>DIA</td>
<td>Diagonal</td>
</tr>
<tr>
<td>BSR</td>
<td>Block Sparse Row</td>
</tr>
<tr>
<td>SSK</td>
<td>Symmetric Skyline</td>
</tr>
<tr>
<td>BSR</td>
<td>Nonsymmetric Skyline</td>
</tr>
<tr>
<td>JAD</td>
<td>Jagged Diagonal</td>
</tr>
</tbody>
</table>

**Note:** CSR = CRS, CCS = CSC, SSK = SKS in some references

**We will focus on COO and CSR!**
Some types of matrices (including graph matrices) contain many zeros. Storing all those zero entries is wasteful. How can we store them so that we avoid storing tons of zeros?

- Python dictionaries (easy, but not efficient)
- Using arrays...?
How can we store a sparse matrix using just arrays? For example:

\[
\begin{pmatrix}
0 & 2 & 0 & 3 \\
1 & 4 & & \\
& & 5 & \\
6 & 7 & \\
\end{pmatrix}
\]

**Idea:** ‘Compressed Sparse Row’ (‘CSR’) format

- Write all non-zero *values* from top-left to bottom-right
- Write down what *column* each value was in
- Write down the index where each *row started*
RowStarts = \begin{pmatrix} 0 & 2 & 4 & 5 & 7 \end{pmatrix} \quad \text{(zero-based)}

Columns = \begin{pmatrix} 1 & 3 & 0 & 1 & 2 & 0 & 3 \end{pmatrix} \quad \text{(zero-based)}

Values = \begin{pmatrix} 2 & 3 & 1 & 4 & 5 & 6 & 7 \end{pmatrix}
$A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
7.0 & 8.0 & 9.0
\end{bmatrix}$

$AA = \begin{bmatrix}
3 & 3 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0
\end{bmatrix}$

- simple
- row-wise
- easy blocked formats
COO

\[ A = \begin{bmatrix}
1 & 0 & 0 & 2 & 0 \\
3 & 4 & 0 & 5 & 0 \\
6 & 0 & 7 & 8 & 9 \\
0 & 0 & 10 & 11 & 0 \\
0 & 0 & 0 & 0 & 12 \\
\end{bmatrix} \]

\[
data = [12.0, 9.0, 7.0, 5.0, 1.0, 2.0, 11.0, 3.0, 6.0, 4.0, 8.0, 10.0]
\]

\[
row = [4, 2, 2, 1, 0, 0, 3, 1, 2, 1, 2, 3]
\]

\[
col = [4, 4, 2, 3, 0, 3, 3, 0, 0, 1, 3, 2]
\]

- simple, often used for entry
CSR

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

\[ data = [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0 ] \]
\[ col = [ 0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4 ] \]
\[ rowptr = [ 0, 2, 5, 9, 11, 12 ] \]

- Length of \( data \) and \( col \) is \( nnz \); length of \( rowptr \) is \( n + 1 \)
- \( rowptr(j) \) gives the index (offset) to the beginning of row \( j \) in \( data \) and \( col \) (\(+1\) due to origin in Fortran)
- no structure, fast row access, slow column access
- related: CSC, Compressed Sparse Column
Sparse Matrix-Vector Multiply

\[ z = Ax, \ A_{m \times n}, \ x_{n \times 1}, \ z_{m \times 1} \]

Input: \( A, x \)
\( z = 0 \)

for \( i = 0 \) to \( m - 1 \)
  for \( \text{col} = A(i,:) \)
    \[ z(i) = z(i) + A(i,\text{col})x(\text{col}) \]
  end
end

- \( O(\text{nnz}) \)
- marches down the rows
- very cheap