WORKING WITH SPARSE MATRICES

David F. Gleich

September 5, 2022

Learning objectives
Learn how to perform basic operations with sparse matrices using the Candyland matrix as an example.

1 INTRO TO CANDYLAND

As we mentioned, there are many real-world problems that involve sparse matrices. In a few lectures, we'll see how discretizations of Laplacian operator on 2d grids will give us sparse matrices. For this class, we are going to continue working with random processes.

The game of Candyland is played on 133 squares. At each turn, a player draws a card from a deck of cards. This determines where they move to for the next turn. There is no interaction with other players (other than sharing the same deck). For our study here, we are going to model the game where we simply draw a random card from the total set at each time, so there is no memory at all in the game. This means that the resulting system is a Markov chain, or a memoryless stochastic process. While there is a great deal of formality we can get into with all of these things, the key thing to remember is that *what happens at each step can just be described by a matrix that tells you the probability of what happens next.*

1.1 THE CANDYLAND MODEL

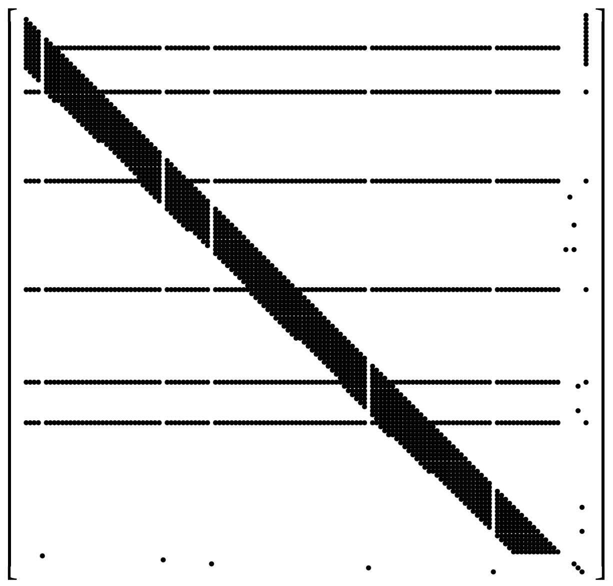
So we are going to create the matrix for Candyland that gives the probabilities of moving between the 133 squares, along with two special squares; one for the start (state 140) and one for the destination (134) . There are also a set of 5 special cases that involve exceptions to the rules (135,136,137,138,139).

In this case, the game of Candyland can be modeled with a 140×140 matrix $\boldsymbol{T}$.[1] If we show the matrix with a small • for each nonzero entry, then it looks like[2]

[1] The data files to recreate $\boldsymbol{T}$ are available on the the course website.

[2] TODO – Double check this one isn't transposed.

$\boldsymbol{T} =$



.

This is clearly sparse as most of the matrix is empty. This is because it's impossible to get between most pairs of squares in Candyland in a single move.

Let $\boldsymbol{T} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \dots & \mathbf{t}_{140} \end{bmatrix}$ be the column-wise partition. Where $\mathbf{t}_j$ describes the probability of ending up at each of the 140 states given that we are in state j. Put another way, $T(i, j)$ is the probability of moving to state i given that you are in state j. Consequently, after one step of the game, the probability that the player is in any state can be read off from $\mathbf{t}_{140}$. This is because the player starts in state 140.

Now, what's the probability of being in any state after two-steps? We can use the matrix to work out this probability:

$$\begin{aligned}&\text{Probability that player is in state } i \text{ after two steps}\\ &\quad = \sum_k \text{Probability that player is in state } k \text{ after one step and moves from } k \text{ to } i.\\ &\quad = \sum_k T(i,k)\mathbf{t}_{140}(k)\end{aligned}$$

If we do this for all i, then we find that

$$\mathbf{p}_2 = \boldsymbol{T}\mathbf{t}_{140}$$

is the probability of the player being in any state after two steps. This is just a matrix-vector operation!

Now to figure out where the player is after any number of steps, we proceed iteratively:

$$\begin{aligned}&\text{Probability that player is in state } i \text{ after three steps}\\ &\quad = \sum_k \text{Probability that player is in state } k \text{ after two steps and moves from } k \text{ to } i.\\ &\quad = \sum_k T(i,k)\mathbf{p}_2(k).\end{aligned}$$

Again, by grouping everything together, we get:

$$\mathbf{p}_3 = \boldsymbol{T}\mathbf{p}_2 = \boldsymbol{T}^2\mathbf{t}_{140}.$$

By induction now, we get that the probability the player is in any state after k steps:

$$\mathbf{p}_k = \boldsymbol{T}^{k-1}\mathbf{t}_{140}.$$

The key point: in order to compute this probability, we only need to compute matrix-vector products with a sparse matrix.

1.2 COMPUTING EXPECTED LENGTH OF A CANDYLAND GAME

The Candyland game ends when then the player is in state 134 in this particular model. Let X be the random variable that is the length of the Candyland game. Then we want to compute the expected value of X. Recall that the expected value of a discrete random variable is:

$$E[X] = \sum\nolimits_{i \text{ where } i \text{ is any possible value of } x} i \cdot (\text{probability that } X = i).$$

The probability that the game ends in 5 steps is[3]

$$[\boldsymbol{T}^4\mathbf{t}_{140}]_{134}.$$

[3] This is the 134 entry in the vector $\boldsymbol{T}^4\mathbf{t}_{140}$.

Hence, the expected length of the Candyland game is:

$$E[X] = \sum_{i=1}^{\infty} i \cdot [\boldsymbol{T}^{i-1}\mathbf{t}_{140}]_{134}.$$

In practice, we can't run this until infinity, even though the game could, in theory, last a very long time. We can compute this via the following algorithm. [4]

```
Create a starting vector $\mathbf{p} = \mathbf{e}_{140}$ because we start in state 140.
EX ← 0
For length = 1 to maximum game length considered
    $\mathbf{p} \leftarrow \boldsymbol{T}\mathbf{p}$
    EX ← EX + length · $\mathbf{p}_{134}$
return EX
```

The key algorithm step is to compute the matrix-vector product $\boldsymbol{T}\mathbf{p}$.

[4] In Julia, the code is

```julia
function candylandlength(T, maxlen)
  n = size(T,1)
  (p = zeros(n))[140] = 1
  ex = 0.0
  for l=1:len
    p = T*p
    ex += length*p[134]
  end
  return ex
end
```

2 SPARSE MATRIX STORAGE: STORING ONLY THE VALID TRANSITIONS FOR CANDYLAND AND PERFORMING A MATRIX-VECTOR PRODUCT

The idea with sparse matrix storage is that we only store the nonzero entries of the matrix. Anything that is not stored is assumed to be zero. This is illustrated in the following figure.

$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Indexed storage

`I`	3	3	5	5	2	1	2	4	4	1
`J`	5	2	6	4	3	3	4	6	3	2
`V`	14	4	4	7	10	13	12	20	9	16

The arrays `I`, `J`, and `V` store the row index, column index, and nonzero value associated with each nonzero entry in the matrix. There are 30 values in the arrays, whereas storing all the entries in the matrix would need 36 values. Although this isn't a particularly large difference, is is less data.

For the matrix $\boldsymbol{T}$ in the Candyland problem, there are 6816 entries in the arrays `I`, `J`, `V` whereas there would be 19600 entries in the matrix $\boldsymbol{T}$ had we stored all the zeros.

We can use this data structure to implement a matrix-vector product. Recall that

$$\mathbf{y} = \boldsymbol{A}\mathbf{x} \text{ means that } y_i = \sum_{j=1,\ldots,n} A_{i,j} x_j \text{ for all } i\ .$$

If $A_{i,j} = 0$ then it plays no role in the final summation and we can write the equivalent expression:

$$\mathbf{y} = \boldsymbol{A}\mathbf{x} \text{ means that } y_i = \sum_{j \text{ where } A_{i,j} \neq 0} A_{i,j} x_j \text{ for all } i\ .$$

This means that an algorithm simply has to implement this accumulation over all *nonzero* entries in the matrix. This is exactly what is stored in the arrays `I`, `J`, `V`.

The algorithm in Julia is:

```julia
function indexed_sparse_matrix_vector_product(x,I,J,V,m,n)
  y = zeros(m)
  for nzi=1:length(I)
    i,j,v = I[nzi], J[nzi], V[nzi]
    y[i] += v*x[j]
  end
  return y
end
```

This algorithm can be translated to many other languages too.

3 ELIMINATING REDUNDANT DATA STORAGE: COMPRESSED SPARSE ROW AND COLUMN FORMATS

The idea with compressed sparse column storage is that some of the information in the full indexed information is redundant *if* we sort all the data by column.[5]

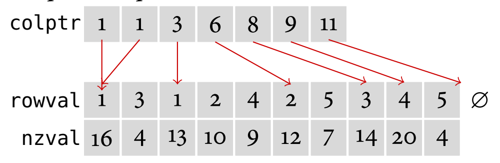
[5] We can also sort by row. That discussion is next.

Sorted index

`J`	2	2	3	3	3	4	4	5	6	6
`I`	1	3	1	2	4	2	5	3	4	5
`V`	16	4	13	10	9	12	7	14	20	4

$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Compressed sparse column



`colptr`	1	1	3	6	8	9	11			
`rowval`	1	3	1	2	4	2	5	3	4	5
`nzval`	16	4	13	10	9	12	7	14	20	4

∅

This figure shows that when the data are sorted by increasing column index. Then there are multiple values with the same column in adjacent entries of the `J` array. We can *compress* these into a list of pointers. This means that we create a new array called `colptr` that stores the starting index for all the entries in `I`, `V` arrays associated with a given column.

$$\text{Entries of column } j \text{ are stored in } \begin{array}{l} \texttt{rowval}[\texttt{colptr}[j]] \ldots \texttt{rowval}[\texttt{colptr}[j+1]-1] \\ \texttt{nzval}[\texttt{colptr}[j]] \ldots \texttt{nzval}[\texttt{colptr}[j+1]-1] \end{array}.$$

This means if $\texttt{colptr}[j] = \texttt{colptr}[j+1]$ then there are no entries in the column. (See the example in column 1.) This

This structure enables efficient iteration over the elements of the matrix for matrix-vector products, just like indexed storage, with only minimal changes to the loop. In Julia, the algorithm is:[6]

[6] This algorithm is especially particular to using 0 based or 1-based indexing.

```
function indexed_sparse_matrix_vector_product(x,colptr,rowval,nzval,m,n)
  y = zeros(m)
  for j=1:n
    for nzi=colptr[j]:colptr[j+1]-1
      i,v = rowval[nzi], nzval[nzi]
      y[i] += v*x[j]
    end
  end
  return y
end
```

Both Julia and Matlab use compressed sparse column formats for their preferred sparse matrix format.

Advantages of compressed sparse column compared with indexed storage.

- less data / memory storage
- allows random access to column data
- allows per-column operations to be more efficient

4 COMPRESSED SPARSE ROW

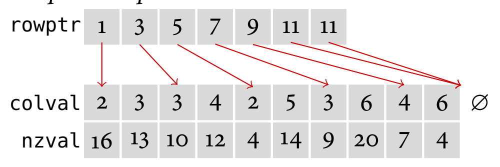
The compressed sparse row format adopts compression of the rows. If we sort row indices from an indexed format, then we can compress them into pointers just as in the compressed sparse column format.

$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Sorted row index

`I`	1	1	2	2	3	3	4	4	5	5
`J`	2	3	3	4	2	5	3	6	4	6
`V`	16	13	10	12	4	14	9	20	7	4

Compressed sparse row



$$\text{Entries of row } i \text{ are stored in } \begin{array}{l} \texttt{colval}[\texttt{rowptr}[i]] \ldots \texttt{colval}[\texttt{rowptr}[j+1]-1] \\ \texttt{nzval}[\texttt{rowptr}[j]] \ldots \texttt{nzval}[\texttt{rowptr}[j+1]-1] \end{array}.$$

If we were to implement the matrix-vector routine for compressed sparse row matrices, however, there is an interesting optimization possible because *all of the updates* to the output vector $\mathbf{y}$ happen in the same index.

```
function indexed_sparse_matrix_vector_product(x,rowptr,colval,nzval,m,n)
  y = zeros(m)
  for i=1:m
    yi = 0.0
    for nzi=rowptr[j]:rowptr[j+1]-1
      j,v = colval[nzi], nzval[nzi]
      yi += v*x[j]
    end
    y[i] = yi
  end
  return y
end
```

An important advantage of this CSR structure is that is is possible to parallelize the sparse matrix vector routine over the rows of the matrix.

5 DISADVANTAGES OF COMPRESSED STORAGE FORMATS

A major disadvantage of compressed sparse column and compressed sparse row formats is that they cannot be easily altered once created.[7] Adding a new nonzero element inside the matrix requires rebuilding the entire array. (Unless it is at the last column!) For this reason, a common paradigm is to use *indexed format* while creating the information for your matrix and then only convert to *compressed sparse* formats when it is time to analyze the matrix and it will be fixed for a reasonably long period of time.

[7] If the number of elements that will change are known ahead of time, then an alternative is to simply insert them into the matrix structure with a "0" placeholder value. It is okay to have *zero* entries in the data structures, and it is okay to alter the values associated with each nonzero element.

Another disadvantage is that we often want to have random access to both rows and columns of a matrix. Compressed sparse column gives efficient random access to columns; compressed sparse row gives efficient random access to rows. But finding all the information for a given row in a compressed sparse column structure involves searching over all the elements. If both random row and random column access are needed, the *easiest* solution is simply to store the matrix both in CSC and CSR formats. This doubles the storage space.[8]

[8] Note that storing a matrix in CSR can be accomplished by storing the transpose in CSC. Likewise, storing a matrix in CSC can be done by storing the *transpose* in CSR.

6 ALTERATIVE FORMATS

Most programming languages have a standard *hash table* or *dictionary* implementation.[9] These allow *arbitrary* key-value pairs to be inserted and give fast access and fast update times. This can be used as a sparse matrix data structure by using the key as an index tuple and the value as the non-zero value. This allows fast insertion and deletion of elements. It does not allow fast random access to rows and columns.

[9] In Julia, these are `Dict` types. In Python, these are also called dictionaries. In C++ the type is `unordered_map`. The matrix type in Julia would be: `DictTupleInt,Int,Float64()` for `Float64` values in the matrix. This idea is implemented in the package `SparseMatrixDicts.jl`

For fast insertion and fast random access to rows and columns, then we can use an array of hash tables.[10]

[10] In Julia, the type would be Vector-DictInt,Float64[].