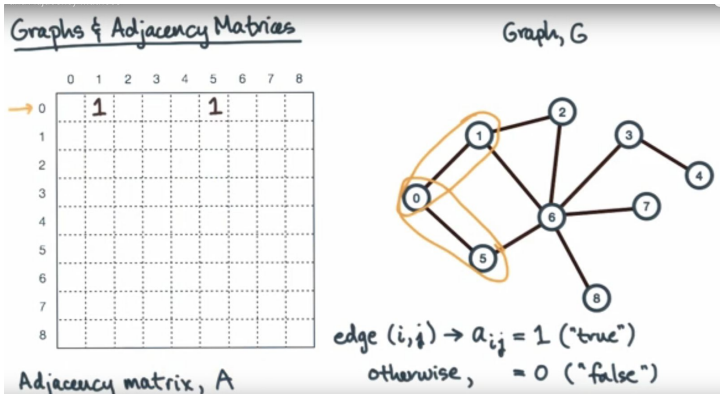
# Lesson 2-6 Distributed BFS

### Graphs and Adjacency Matrices

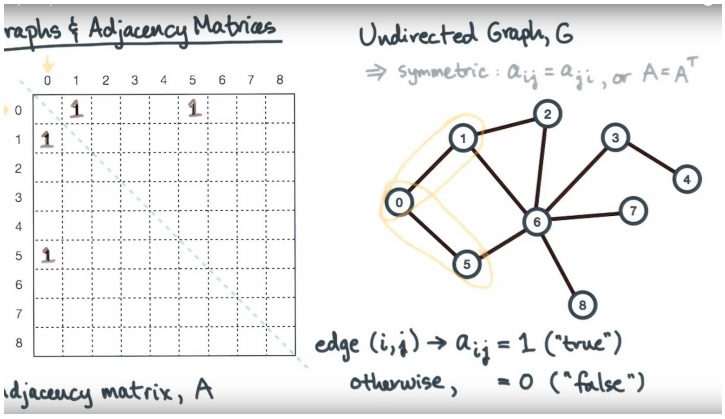
It is helpful to look at a graph as an adjacency matrix.

To do this:

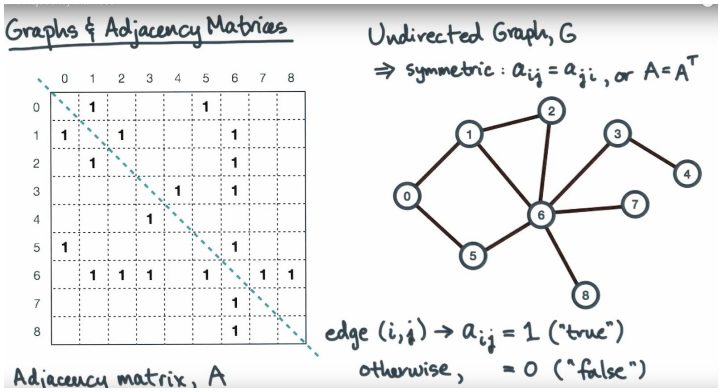
1. Give each vertex an integer label.
2. Then create a matrix, A, to represent the graph.  A row for every edge, and a column for every vertex.
3. For each edge, put a ‘1’ in the corresponding row, column. For the vertex ‘0’, it has two edges attached to it. One of the edges is going to vertex ‘1’ and the other to vertex ‘5’.




4. Undirected graphs have symmetric matrix. So put a ‘1’ in the corresponding columns.




5. The final graph



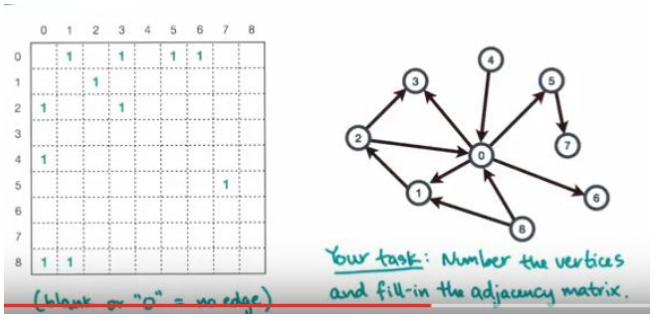

For any undirected graph ‘G’,
Assume:

n = number of vertices
m = number of edges

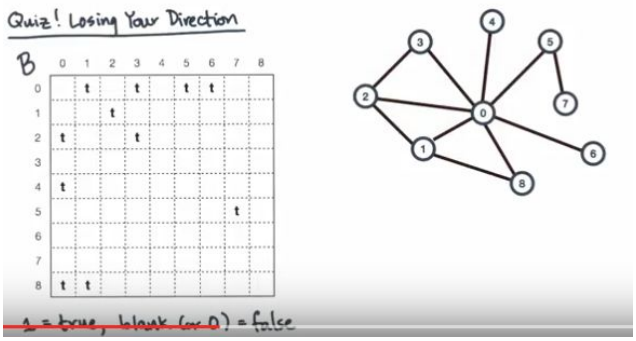
Its adjacency matrix will be nxn.
The number of nonzeros: nnz(A) = 2m (# of nonzeros)

**The Adjacency Matrix for a DIRECTED Graph**

Arbitrarily number the graph.
Then record the OUTGOING edges of each vertex.



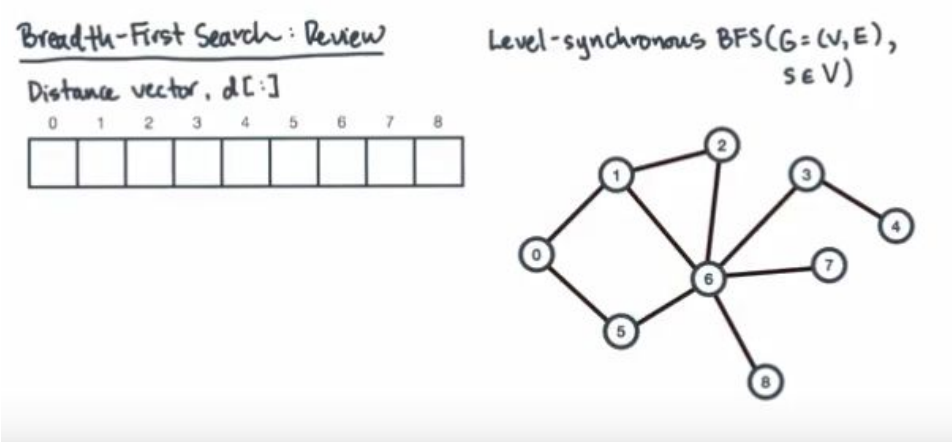
Now treat the matrix as a boolean matrix:
1 = True, 0=False




What is the undirected graph for this directed graph?

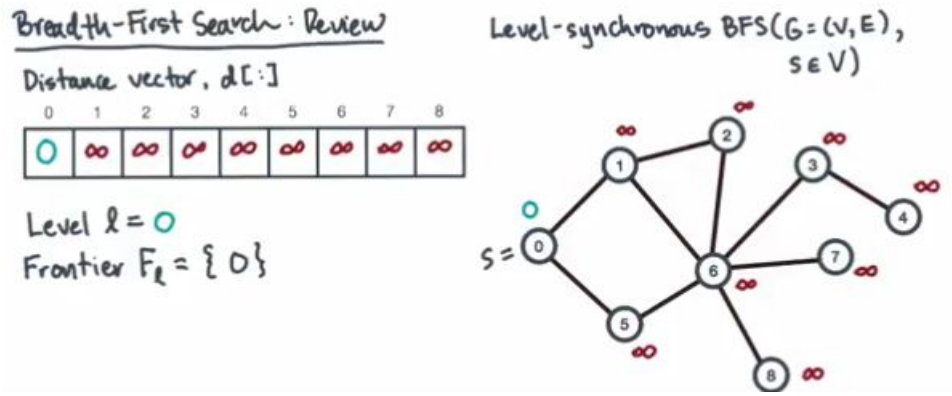
The logical ‘or’ of B and its transpose.

**Breadth-First Search: Review**




First: calculate the minimum distance of each vertex from ‘s’.
Let vertex ‘0’ be ‘s’.

‘s’ is 0 distance from itself, all other vertices are infinite distance from ‘s’.

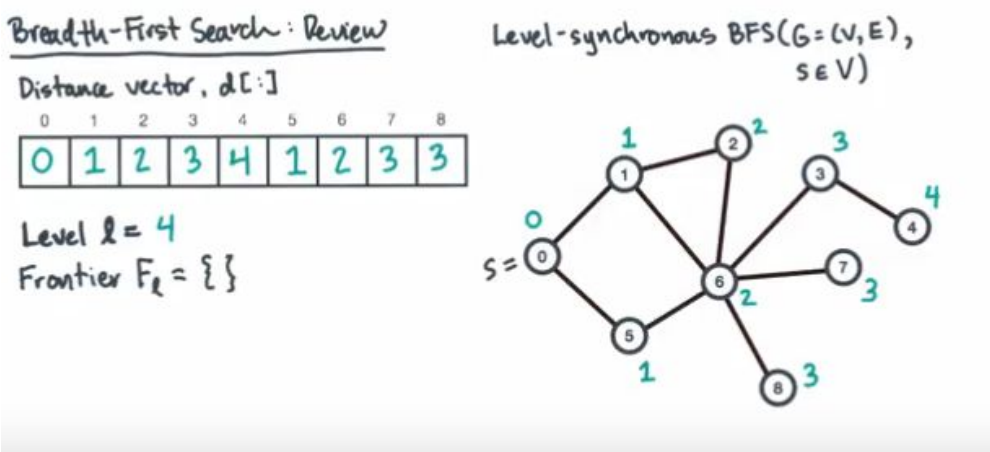


For level = 0, the Frontier $F_l$= {0}. For each level, it is just the distance of the vertices from s.

So now we need to visit all of the frontiers neighbors, in this case the neighbors of the frontier are vertices 1 and 5. Their distances are updated with the current distance + 1. (in this case 0 + 1 = 1)

These visited neighbors now become the new frontier. The level is 1, so update all neighbors to distances 1+1 = 2.

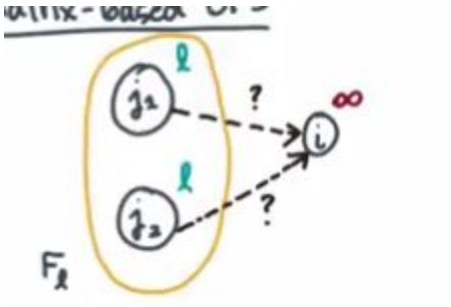
Repeat for all nodes.
The final result:




The cost of the algorithm is: O(m + n) n = vertices, m = edges

**Matrix-based BFS**

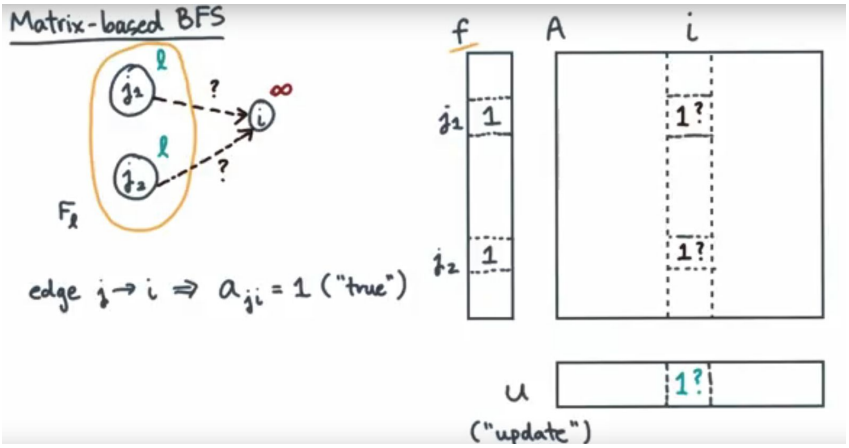
Now - translate the BFS algorithm into the language of Matrices.




Is there an edge from the frontier to i? If there is, the distance should be updated.

If there is an edge from j → i, then the adjacency matrix should have a '1' (or true) at $a_{ji}$.

Consider the graph as a matrix A and the frontier as a boolean vector 'f'.
To determine if there is an edge from i to the frontier, there needs to be an i at the corresponding vertices. To record these edges, mark them in a boolean vector called 'u' (for update).

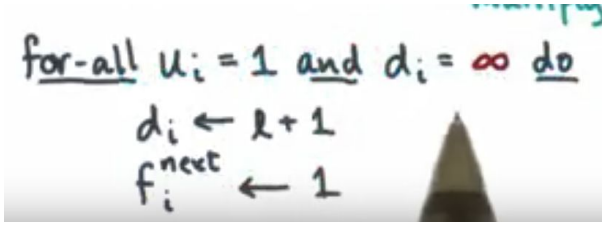



This can be said as:
Update i if any vertex j  is in the frontier and points to i.

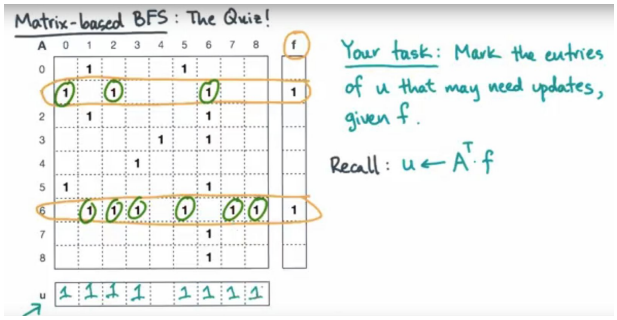
$$u[i] \leftarrow \bigvee_{j=1}^{n} (f_j \wedge a_{ji})$$

(V is logical ‘or’ and the up carrot is a logical ‘and’)

For a sparse graph, the vector and the matrix can be maintained with sparse data structures. to go from the update vector to the updated distances,




**MATRIX BASED BFS THE QUIZ**



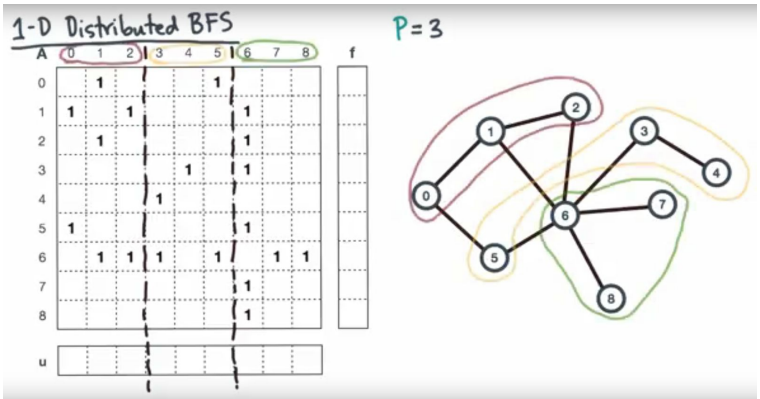

The example of filling in the update vector. Use the the matrix-vector product.

**1-D Distributed BFS**

The matrix gives you an easy way for distributing the BFS.

To do this:

Divide the number of columns into equal sections for each processor. The column partitioning corresponds to the partitioning of the vertices.




The update vector will be partitioned to each process, but the frontier vector will need to be replicated on each process.

With each update, you will need to replicate the frontier again.

Use all-to-all

**The Distributed 1-D Algorithm**

1. partition columns of A and entries of u.
2. Computer $u \leftarrow A^T f$
3. Locally update the local distances
4. Identify local vertices of the next frontier
5. To an all-to-all exchange of the frontier

The all-to-all is the only communication step, what is the cost?

The 1-D costs scale linearly. What is the cost when it is 2-D? Square root of p (p is the number of processors).