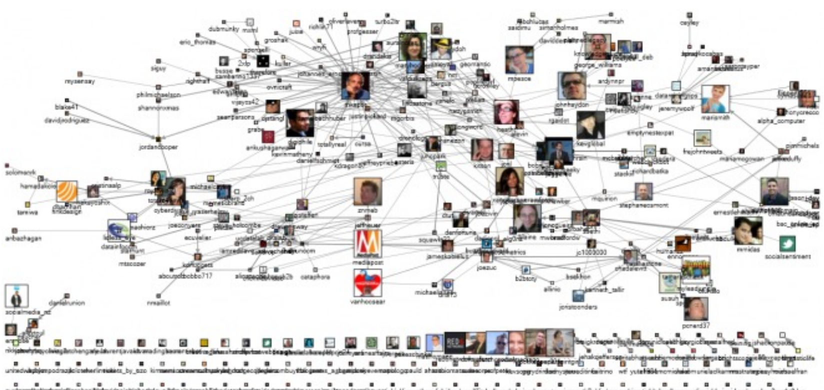
Parallel Patterns: Graph Search

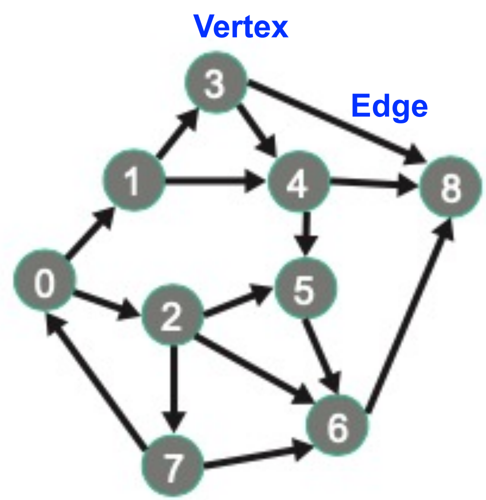
Prof. Seokin Hong

Graph

- A **graph data structure** represents the **relations** between **entries**
  - Example
    - **Social network:** entities are users and relations are connections between users
    - **Driving direction app:** entities are locations and the relations are the roadways between them

- Relations are bi-directional or directional



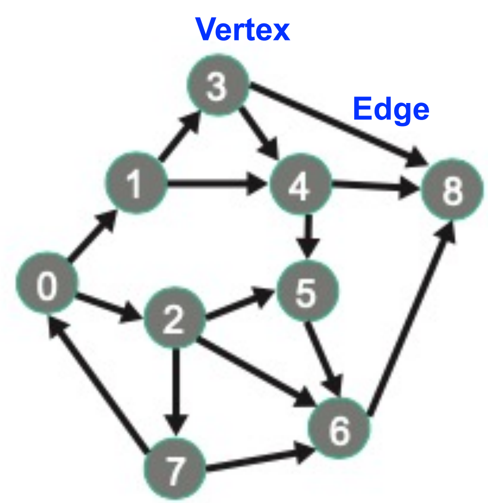


Graph with directional edges

Representation of a Graph

- **Adjacency matrix** can be used to represent a graph
  - Assign an unique number to each vertex
  - When there is an edge going from vertex i to vertex j, the value of element A[i][[j] is 1. Otherwise, it is 0.

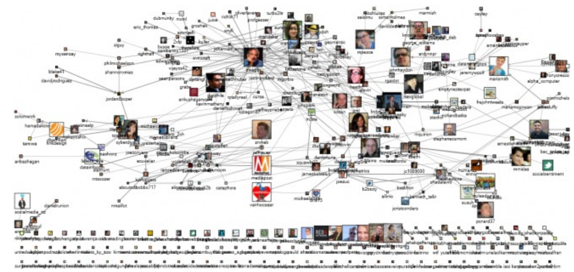
	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									



Graph with directional edges

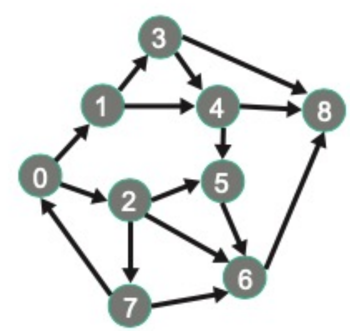
Sparsely-connected Graph

- **Usually, graph is sparsely connected**
  - If the number of vertex is N, the average number of outgoing edges from each vertex is much smaller than N-1
  - Many real-world graphs are sparsely connected
    - Ex) a social network such as Facebook



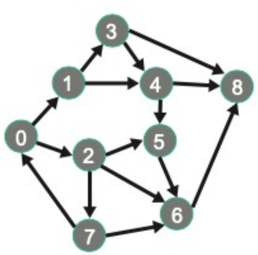
- So, the number of non-zero elements in the adjacency matrix is much smaller than the total number of elements
- Using sparse matrix representation (e.g., CSR) to represent a sparsely-connected graph drastically reduce the amount of storage and the number of wasted operations

	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									



Sparse matrix representation of sparsely-connected Graph

- Row pointer array gives starting location of non-zero elements (**edges**)
  - Example
    - edges[3] gives the starting location (=7) of non-zero elements in row 3 (vertex 3)
    - edges[4] gives the starting location (=9) of non-zero elements in row 4 (vertex 4)
    - So, we expect to find non-zero elements for row 3 in data[7] and data[8] and the column indices for these elements in destination[7] and destination[8]
- Column index array gives **destination** of each edge
  - Ex) the destination of the two edges for source vertex 3 are destination[7]=4 and destination[8]=8
- Data array can be used to store additional information about the relationship
  - Ex) distance between two locations or date when two social network users became connected

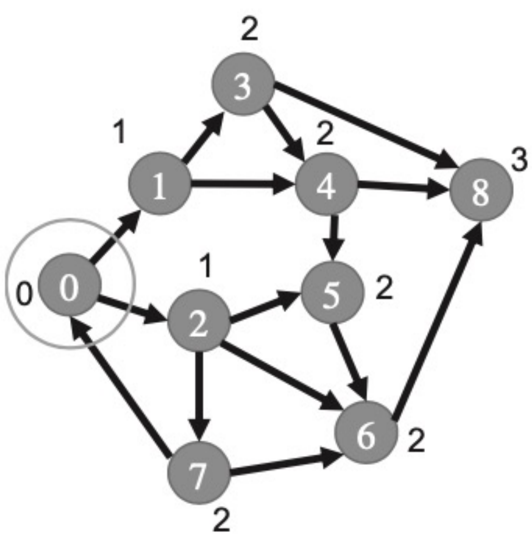


	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

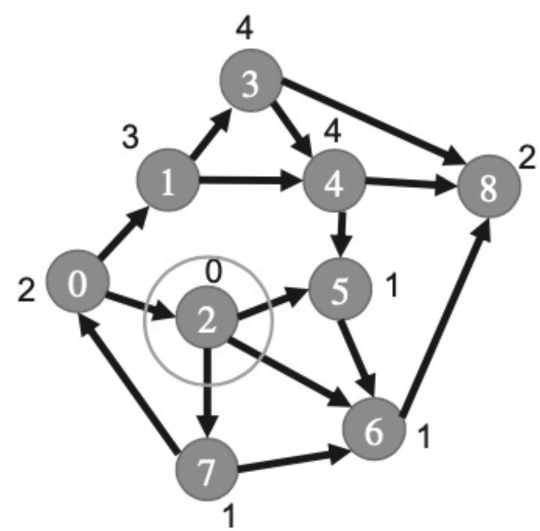
Non-zero elements data[15]	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Column indices destination[15]	1	2	3	4	5	6	7	4	8	5	8	6	8	0	6
Row pointers edges[10]	0	2	4	7	9	11	12	13	15	15					

Breadth-First Search (BFS)

- BFS is a graph traversing algorithm that traverses a graph layerwise thus exploring the neighbor nodes (nodes which are directly connected to source node).
- BFS is often used to discover the shortest path between two vertices
    - Each vertex is labeled with the smallest number of edges that one needs to traverse in order to go from the source to the vertex



Desired BFS result with vertex 0 as the source

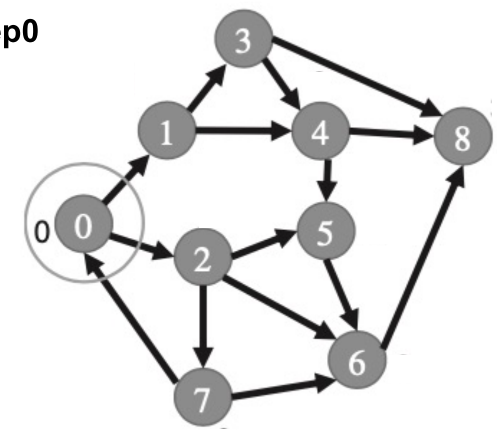


Desired BFS result with vertex 0 as the source

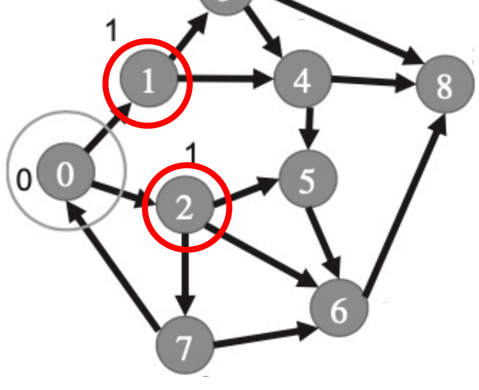
Breadth-first Search (BFS)

- Example of labeling vertices with BFS : source is vertex 0

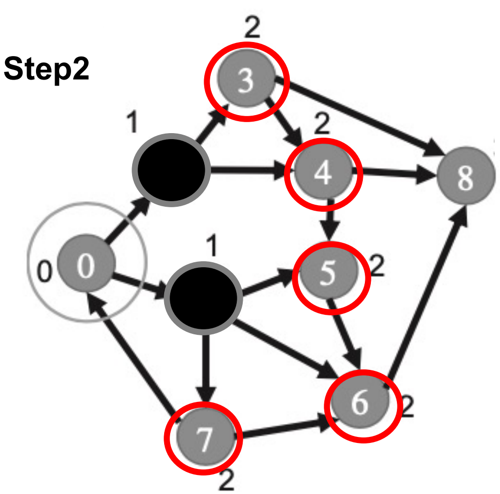
Step0



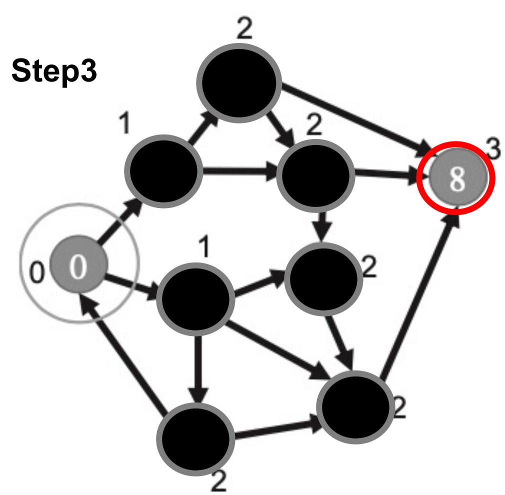
Step1



Step2

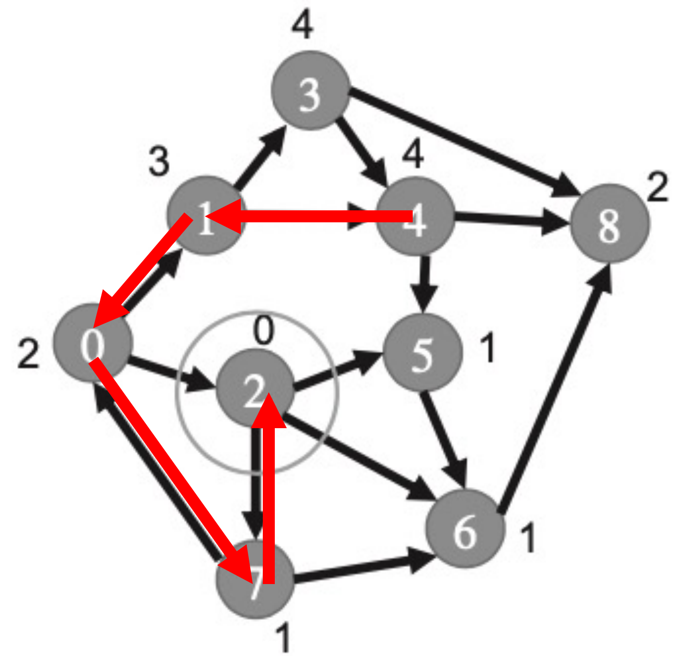


Step3



Breadth-first Search (BFS)

- **Finding shortest path**
  - Start from destination vertex and trace back to the source
  - Select a vertex that has smallest number of edges



Sequential BFS function

- The graph is **represented in the CSR format**
- **The function receives**
  - the index of the source vertex
  - edges array (row pointer array of CSR)
  - destination array (column index array of CSR)
  - label array whose elements are used to store the visit status information for the vertices

- **Before search,**
  - Source vertex is labeled to 0, indicating it is a level 0 vertex
  - All other label elements are initialized to -1, indicating that they are not visited

- **At the end of the searh,**
  - all vertices reachable from the source are labeled with a positive number
  - If the label of a vertex is -1, it means that the vertex is unreachable from the source

Implementation of Sequential BFS function

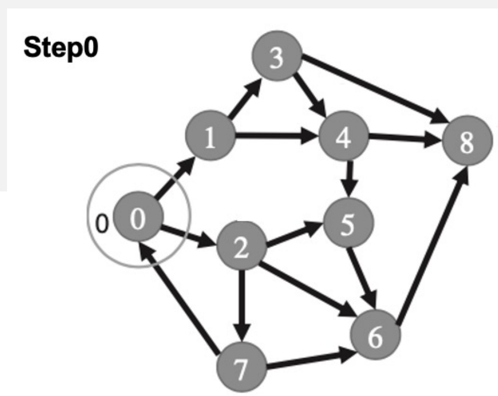
```
void BFS_sequential(int source, int *edges, int *dest, int *label)
{
   int frontier[2][MAX_FRONTIER_SIZE];    //two frontier arrays
   int *c_frontier = &frontier[0];        /*c_frontier array stores the frontier vertices that are
                                            being discovered in the current iteration */
   int c_frontier_tail =0;                /*stores the index of position at which a newly discovered
                                            frontier vertex can be added in c_frontier array*/
   int * p_frontier = &frontier[1];       /*p_frontier array stores the frontier vertices discovered in the
                                            previous iteration*/
   int p_frontier_tail =0;      //number of elements that have been inserted into the p_frontier

   insert_frontier(source, p_frontier, &p_frontier_tail);   //place source vertex into p_frontier and
                                                              increment p_frontier_tail to 1
   label[source] =0;      //label of a source vertex is initialized to 0
   …..
```

A frontier : vertices that are not yet visited but are adjacent to a visited vertex and visit a vertex only when it is in the frontier

Implementation of Sequential BFS function

```
.....
while (p_frontier_tail >0) {
    for(int f=0; f<p_frontier_tail; f++){                //visit all previous frontier vertices
        c_vertex= p_frontier[f];                         //pick up one of the previous frontier vertex
        for (int i= edges[c_vertex]; i<edges[c_vertex+1];i++) {  //for all its edges
            if(label[dest[i]] == -1){                    //the dest vertex has not been visited
                insert_frontier(dest[i], c_frontier, &c_frontier_tail); //insert discovered vertex into c_frontier
                label[dest[i]] = label[c_vertex]+1;
            }
        }
        int temp= c_frontier; c_frontier = p_frontier; p_frontier = temp; //swap previous and current frontier
        p_frontier_tail = c_frontier_tail;
        c_frontier_tail = 0;
}
```

Non-zero elements data[15]	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Column indices destination[15]	1	2	3	4	5	6	7	4	8	5	8	6	8	0	6
Row pointers edges[10]	0	2	4	7	9	11	12	13	15	15					



label

c_frontier

p_frontier

Parallel BFS Function

- Parallelize each iteration of the while-loop
  - Multiple threads collaboratively process the previous frontier array and assemble the current frontier array
  - How?
    - Assign a section of the previous frontier array to each thread block

Parallelize

```
while (p_frontier_tail >0) {
  for(int f=0; f<p_frontier_tail; f++){                    //visit all previous frontier vertices
      c_vertex= p_frontier[f];                             //pick up one of the previous frontier verte
       for (int i= edges[c_vertex]; i<edges[c_vertex+1];i++) {  //for all its edges
           if(label[dest[i]] == -1){                     //the dest vertex has not been visited
               insert_frontier(dest[i], c_frontier, &c_frontier_tail); //insert discovered vertex into c_fronti
```

Implementation of Parallel BFS Function

- Host code

```
void BFS_host(unsigned int source, unsigned int*edge, unsigned int *dest, unsigned int *label)
{
    //allocate edges_d, dest_d, label_d, and visited_d in device global memory
    //copy edges, dest, and label to device global memory
    // allocate frontier_d, c_frontier_tail_d, p_frontier_tail_d in device global memory
    …..
   unsigned int *c_frontier_d = &frontier_d[0];                          //c_frontier_d point the first half
   unsigned int *p_frontier_d = &frontier_d[MAX_FRONTIER_SIZE];    //c_frontier_d point the second half

   //launch a simple kernel to initialize the following in the device global memory
   //initialize all visited_d elelments to 0 except source to 1
      //*c_frontier_tail_d = 0;
      //p_frontier_d[0] = source;
      //*p_frontier_tail_d = 1;
      //label_d[source] =0;
  p_frontier_tail = 1;
  ….
```

Implementation of Parallel BFS Function

- Host code (cont'd)

```
….
  while(p_frontier_tail>0) {
      int num_blocks = ceil(p_frontier_tail/float(BLOCK_SIZE));

      BFS_Bqueue_kernel<<num_blocks, BLOCK_SIZE)>>>(p_frontier_d, p_frontier_tail_d,
c_frontier_d, c_frontier_tail_d, edges_d, dest_d, label_d, visited_d);

     //use cudaMemcpy to read the *c_frontier_tail value back to host
     // and assign it to p_frontier_tail for the while-loop condition test

      int* temp = c_frontier_d; c_frontier_d=p_frontier_d; p_frontier_d=temp; //swap the roles
      // launch a simple kernel to set *p_frontier_tail_d = *c_frontier_tail_d; *c_frontiler_tail_d=0;
     }
}
```

Implementation of Parallel BFS function

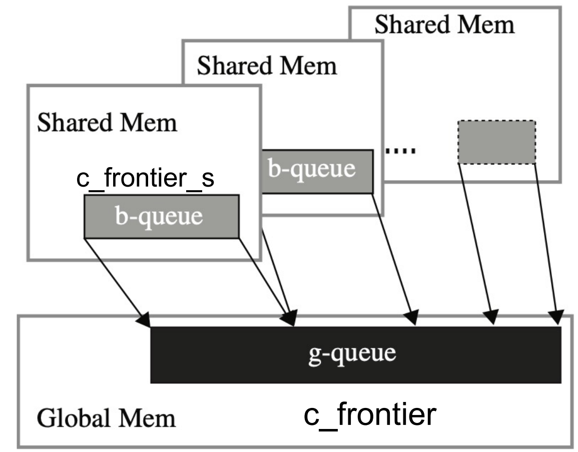
- Kernel

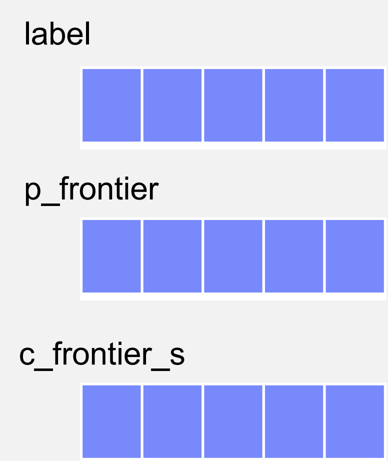
```
__global__ void BFS_Bqueue_kernel(unsigned int *p_frontier, unsigned int *p_frontier_tail,
unsigned int *c_frontier, unsigned int *c_frontier_tail, unsigned int *edges, unsigned int *dest,
unsigned int *label, unsigned int* visited)
{
    __shared__ unsigned int c_frontier_s[BLOCK_QUEUE_SIZE];
    __shared__ unsigned int c_frontier_tail_s, our_c_frontier_tail;

    if(threadIdx.x == 0) c_frontier_tail_s = 0;
    __syncthreads();

    const unsigned int tid = blockIdx.x*blockDim.x +threadIdx.x;
    …..
```

Implementation of Parallel BFS function

- Kernel (cont'd)

```
….
if (tid < *p_frontier_tail){
    const unsigned int my_vertex = p_frontier[tid];
    for(unsigned int i = edges[my_vertex]; i< edges[my_vertex+1] ; i++)
    {
        const unsigned int was_visited = atomicExch(&(visited[dest[i]]),1);
        if(!was_visited){
            label[dest[i]] = label[my_vertex]+1;
            const unsigned int my_tail = atomicAdd(&c_frontier_tail_s,1);
            if(my_tail < BLOCK_QUEUE_SIZE) { // add discovered vertex to the local queue (shared memory)
                c_frontier_s[my_tail]= dest[i];
            } else {                          //if full, add it to the global queue directly (global memory)
                c_frontier_tail_s =BLOCK_QUEUE_SIZE;
                const unsigned int my_global_tail = atomicAdd(c_frontier_tail,1);
                c_frontier[my_global_tail] = dest[i];
            }
        }
    }
}
….
```





Implementation of Parallel BFS function

- Kernel (cont'd)

```
 ….
__syncthreads();
    if(threadIdx.x ==0){
            our_c_frontier_tail = atomicAdd(c_frontier_tail, c_frontier_tail_s);
   }
   __syncthreads();
   for(unsigned int i= threadIdx.x; i<c_frontier_tail_s; i+=blockDim.x){
          c_frontier[our_c_frontier_tail +i]=c_frontier_s[i];
  }
}
```