CPSC 4240/5240: Parallel Programming Techniques

Lecture 13: CSR and Graphs

Lecturer: Quanquan C. Liu

Spring 2026

What is CSR?

- CSR represents a sparse matrix using **three arrays**:
  - **Values (VAL):** Stores all nonzero elements row-wise.
  - **Column Indices (COL):** Stores the column indices corresponding to each nonzero element.
  - **Row Pointers (ROW_PTR):** Marks the **starting index** of each row in the VAL and COL arrays.

Example CSR Format

- Consider the following **4×5 sparse matrix**:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

- Let’s convert to CSR step by step

Example CSR Format

- Create the VAL array
- Store all **nonzero elements** row by row:

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array
- For each nonzero element in VAL, store its **column index**:

COL=[2,4,0,3,1,4]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array
- For each nonzero element in VAL, store its **column index**:

COL=[2,4,0,3,1,4]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

- **3** is in col **2, 4** is in col **4** etc.

Example CSR Format

VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

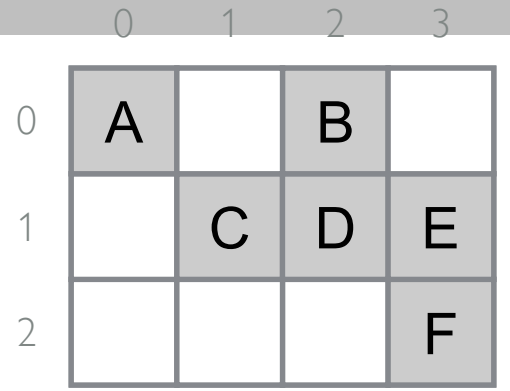
VAL = [3,4,5,7,8,6]

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

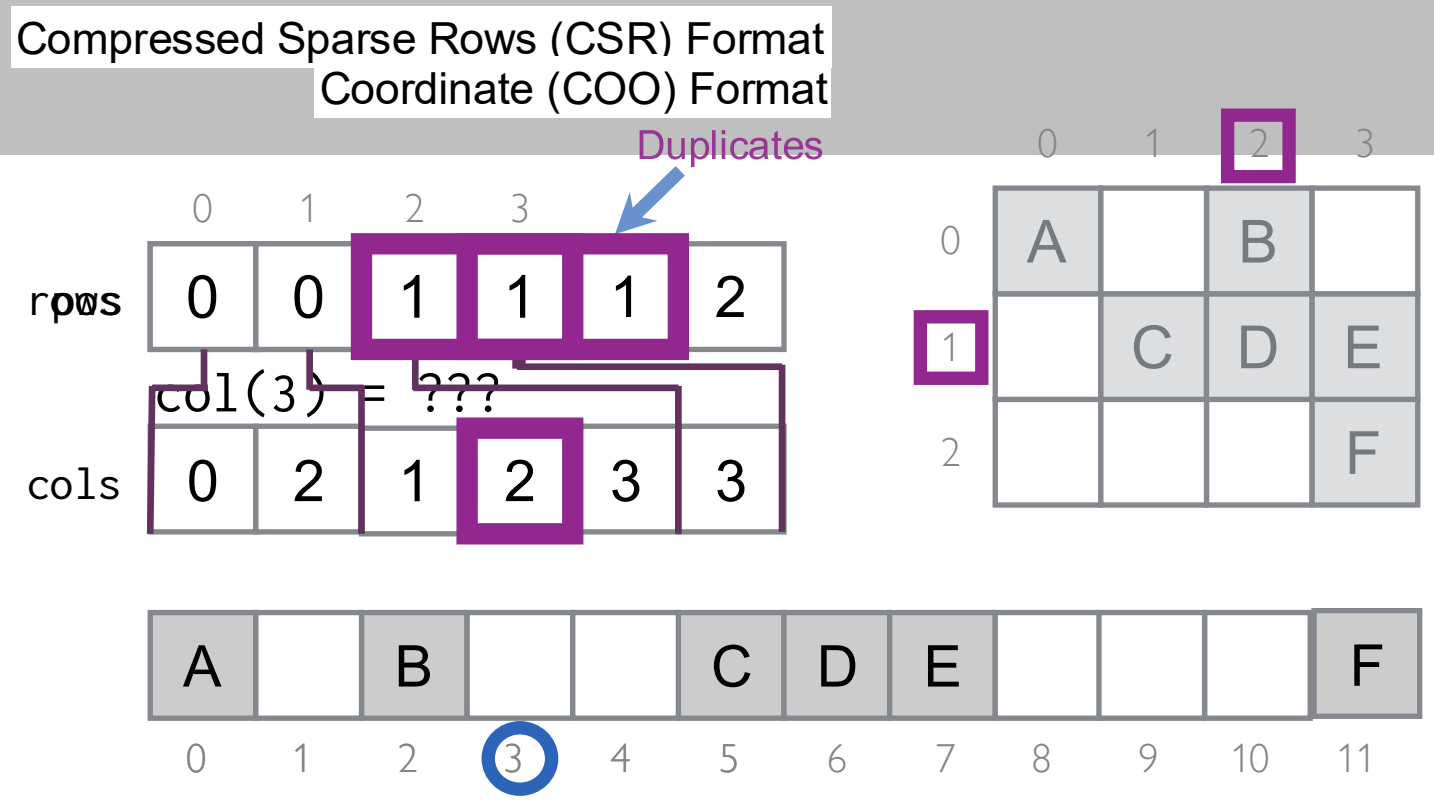
ROW_PTR = [0,2,2,4,6]

Data Representations in CSR



Slide source: Courtesy of the 6.106 Lecturers 2022

Data Representations in CSR



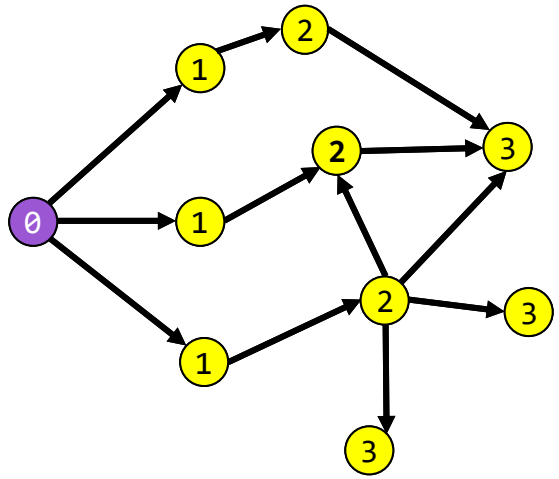
Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search

Example use of CSR in Graph Algorithms

Breadth-First Search

- Problem: Given a directed unweighted graph and a source vertex, find the depth of each vertex from the source

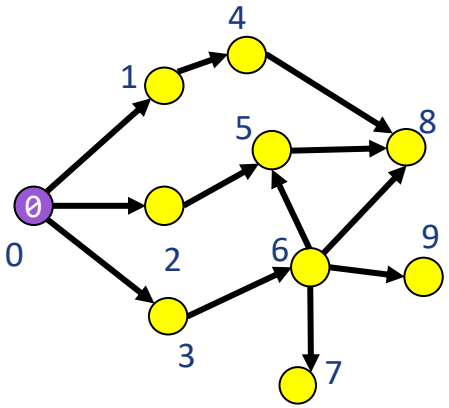


Powerful primitive in many other graph algorithms

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential

BFS: Sequential Implementation



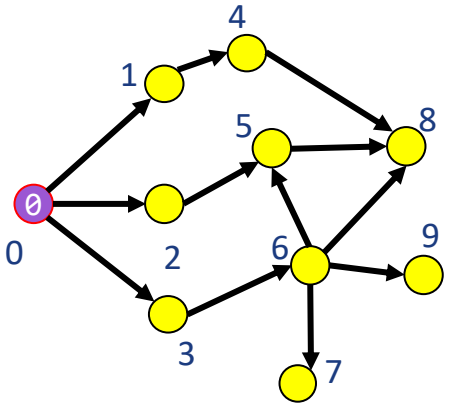
	0	1	2	3	4	5	6	7	8	9
depth	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
Q	0									

- Initialize depth to -1
- Initialize queue with source vertex

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential

BFS: Sequential Implementation



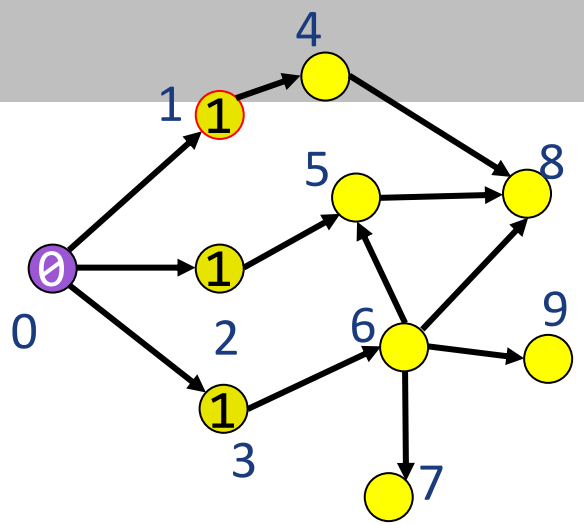
depth	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9

Q	0									

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



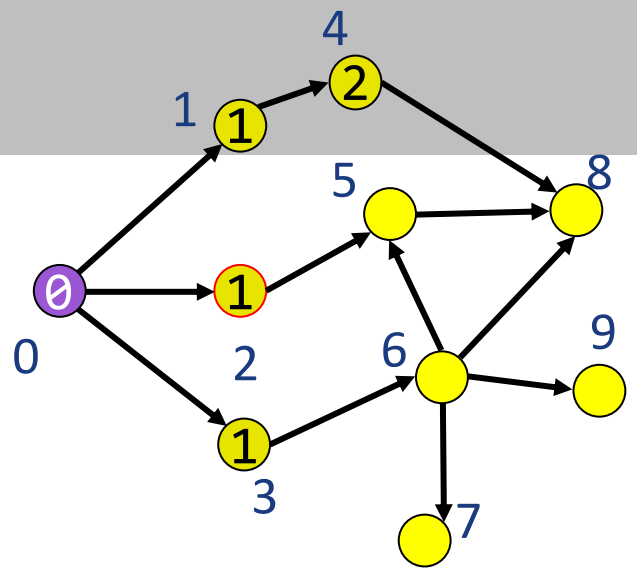
depth	0	1	1	1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9

Q	~~0~~	1	2	3						

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



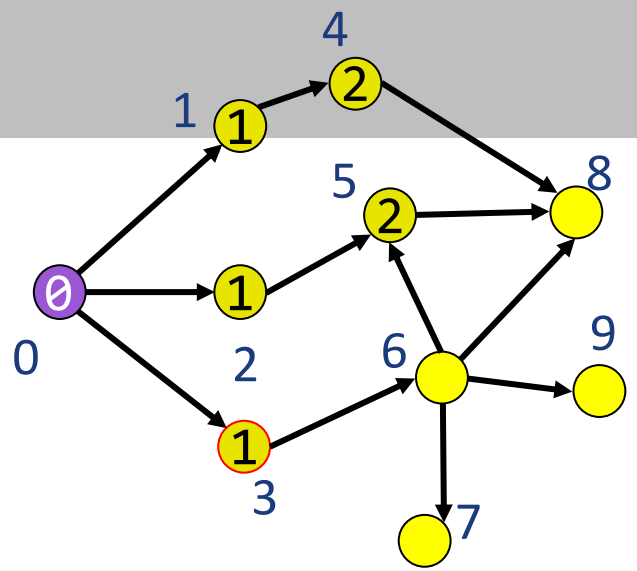
depth	0	1	1	1	2	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9

Q	~~1~~	~~2~~	2	3	4					

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential

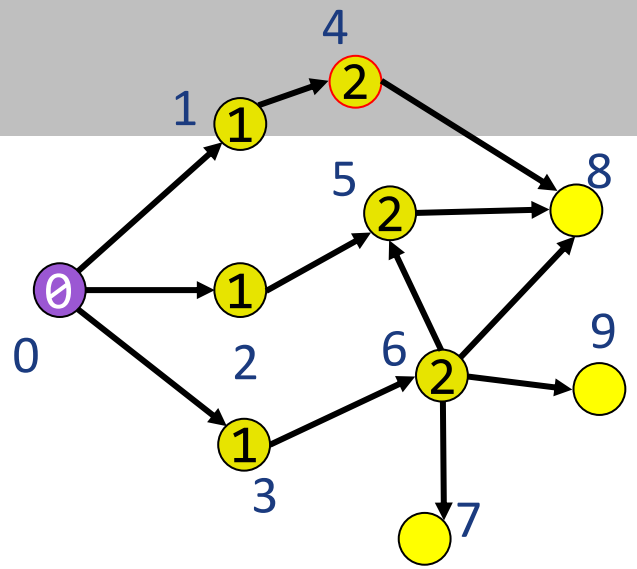


depth	0	1	1	1	2	2	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9
Q	X	X	X	3	4	5				

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential

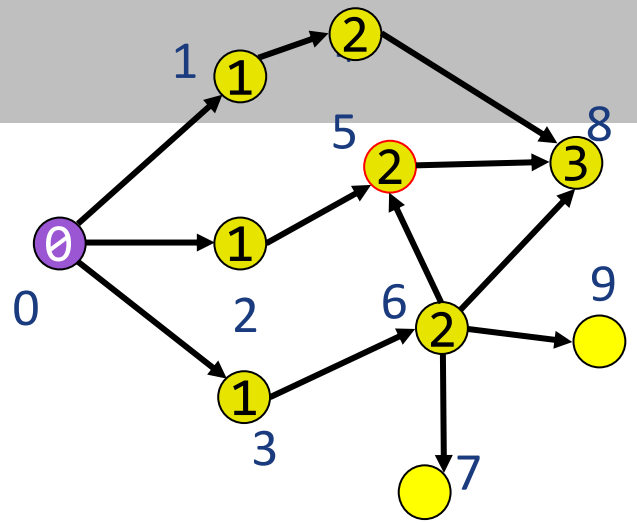


	0	1	2	3	4	5	6	7	8	9
depth	0	1	1	1	2	2	2	-1	-1	-1
Q	✗	✗	✗	✗	4	5	6			

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



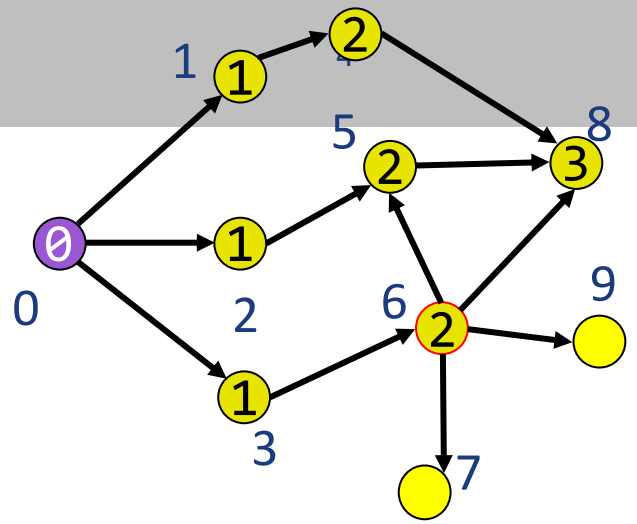
depth	0	1	2	3	4	5	6	7	8	9
	0	1	1	1	2	2	2	-1	3	-1

Q						5	6	8		

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



depth

0	1	2	3	4	5	6	7	8	9
0	1	1	1	2	2	2	-1	3	-1

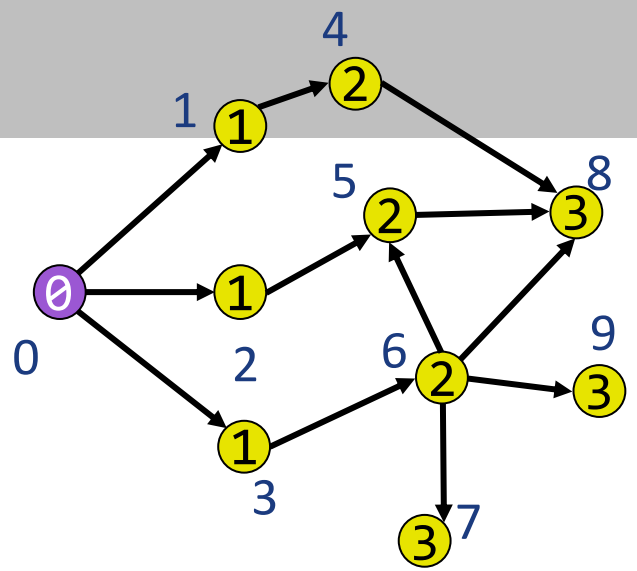
Q

✗	✗	✗	✗	✗	✗	6	8		

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



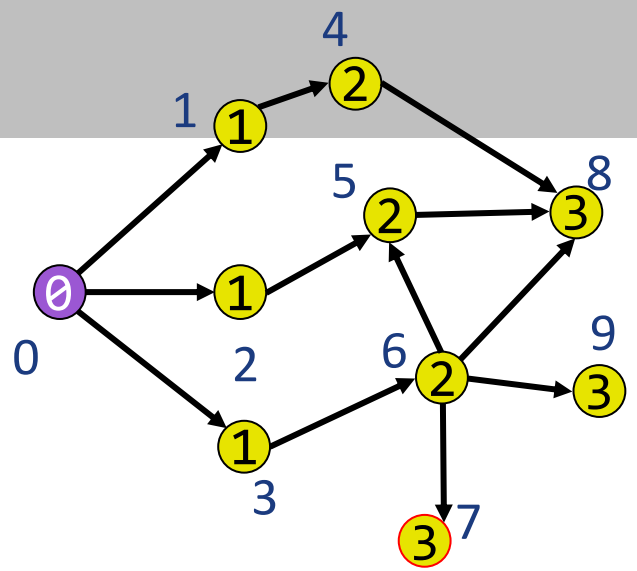
depth	0	1	1	1	2	2	2	3	3	3
	0	1	2	3	4	5	6	7	8	9

Q	✗	✗	✗	✗	✗	✗	✗	8	7	9

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential

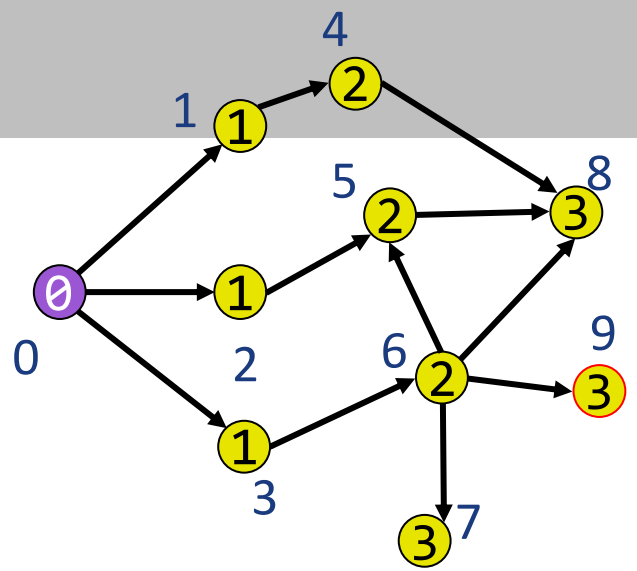


depth	0	1	1	1	2	2	2	3	3	3
	0	1	2	3	4	5	6	7	8	9
Q	X	X	X	X	X	X	X	X	7	9

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



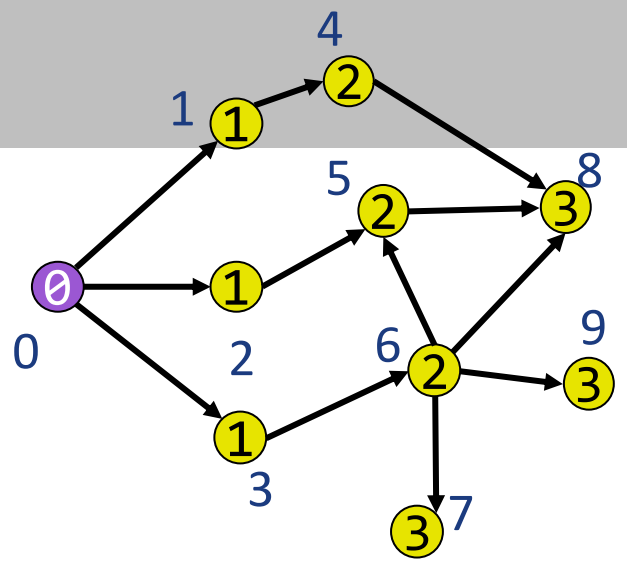
depth	0	1	1	1	2	2	2	3	3	3
	0	1	2	3	4	5	6	7	8	9

Q										
	✗	✗	✗	✗	✗	✗	✗	✗	✗	9

- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Sequential



depth

0	1	1	1	2	2	2	3	3	3
0	1	2	3	4	5	6	7	8	9

Q

X	X	X	X	X	X	X	X	X	X

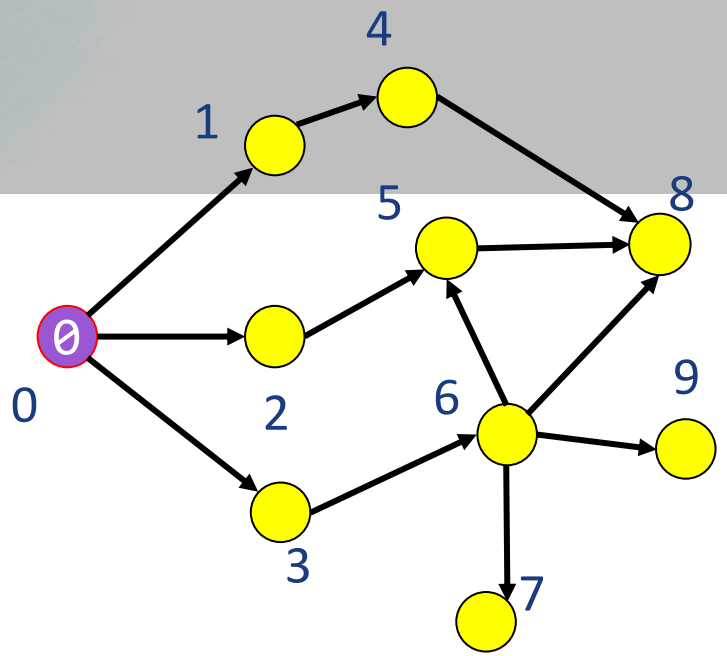
- While queue is not empty, dequeue a vertex
  - Update all the neighbors of the vertex ($depth_j = depth_i + 1$) if $depth_j$ is -1
  - Add all updated neighbors to queue

Slide source: Courtesy of the 6.106 Lecturers 2022

How do we make it parallel?

Ideas? Which of these operations can be done in parallel?

Breadth-First Search: Parallel



depth

0	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9

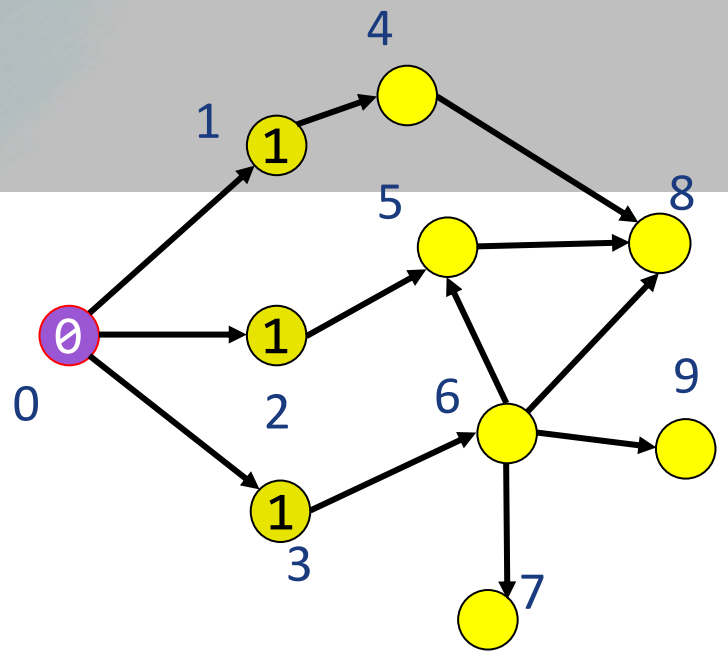
$frontier_{in}$	0						

$frontier_{out}$							

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



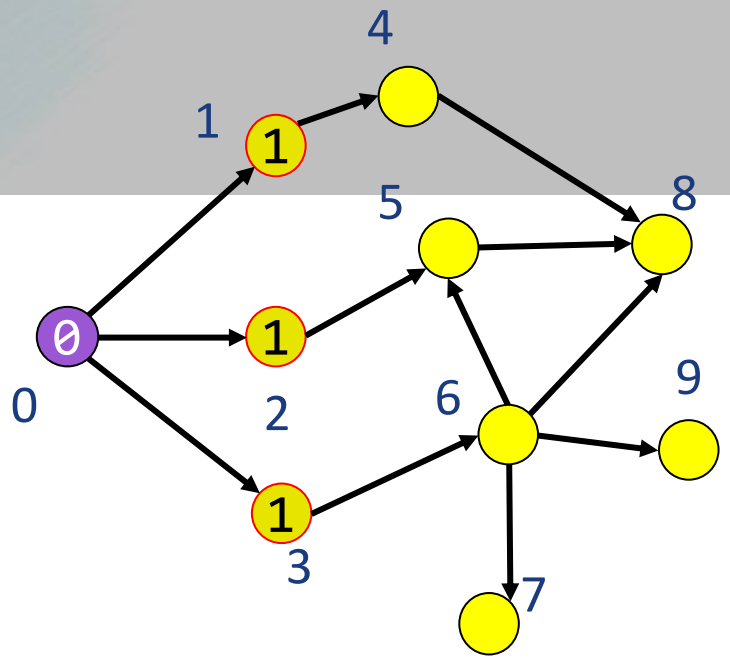
depth	0	1	1	1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9

$frontier_{in}$	0						
$frontier_{out}$	1	2	3				

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



depth	0	1	1	1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9

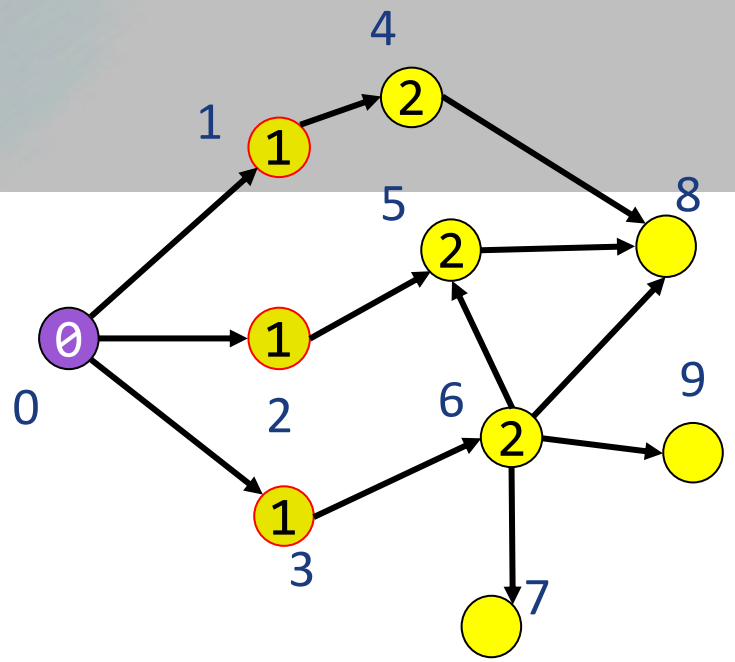
$frontier_{in}$	1	2	3				

$frontier_{out}$							

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



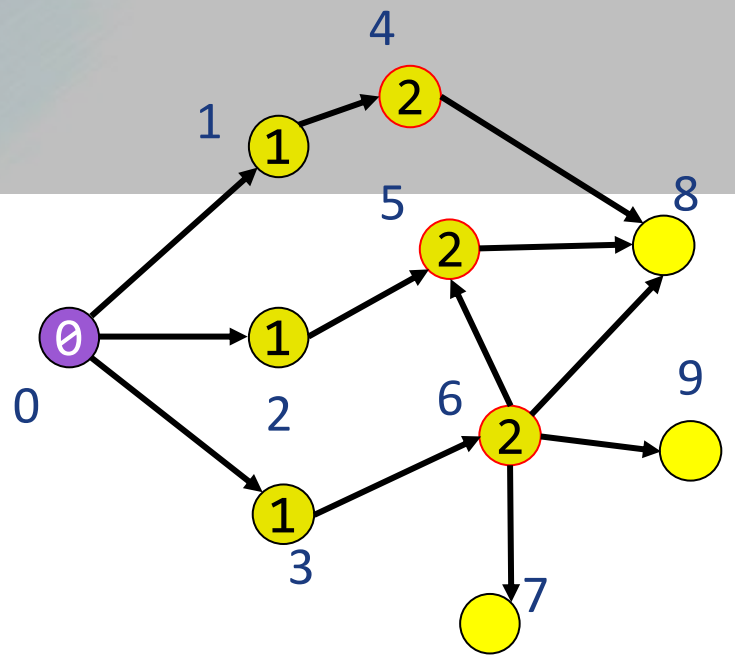
	0	1	2	3	4	5	6	7	8	9
depth	0	1	1	1	2	2	2	-1	-1	-1

$frontier_{in}$	1	2	3				
$frontier_{out}$	4	5	6				

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



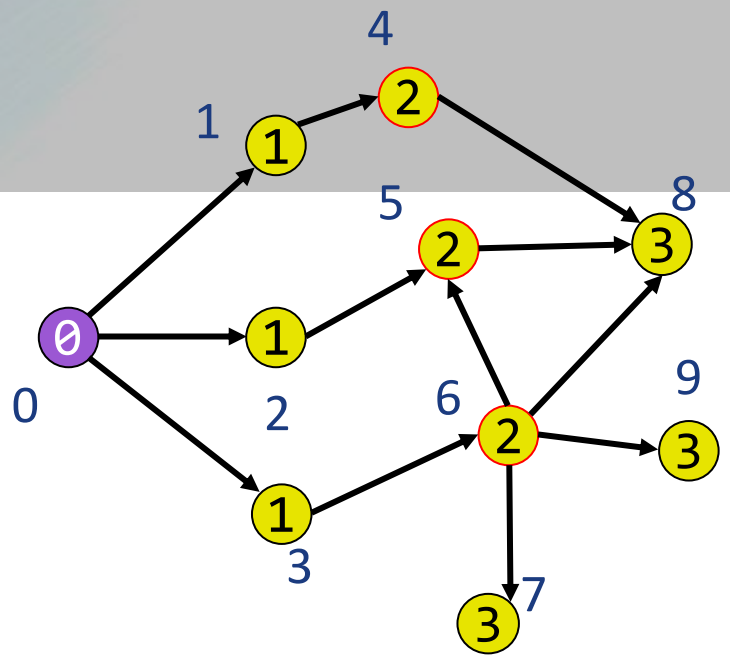
	0	1	2	3	4	5	6	7	8	9
depth	0	1	1	1	2	2	2	-1	-1	-1

$frontier_{in}$	4	5	6				
$frontier_{out}$							

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



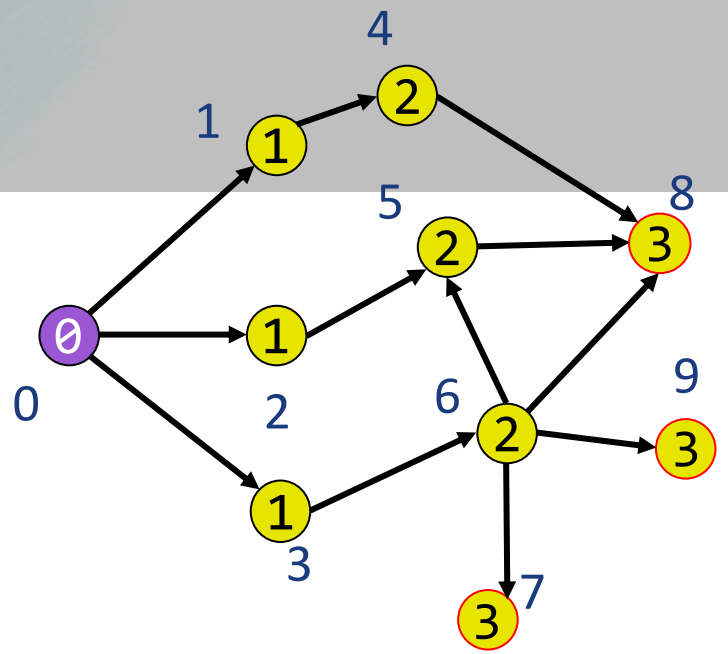
depth	0	1	1	1	2	2	2	3	3	3
	0	1	2	3	4	5	6	7	8	9

$frontier_{in}$	4	5	6				
$frontier_{out}$	8	9	7				

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



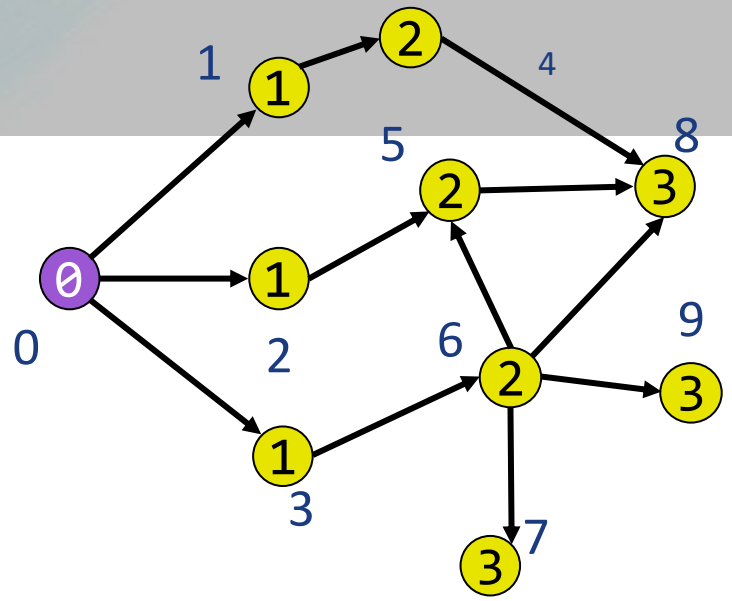
depth	0	1	2	3	4	5	6	7	8	9
	0	1	1	1	2	2	2	3	3	3

$frontier_{in}$	8	9	7				
$frontier_{out}$							

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Breadth-First Search: Parallel



depth	0	1	1	1	2	2	2	3	3	3
	0	1	2	3	4	5	6	7	8	9

$frontier_{in}$							
$frontier_{out}$							

- While $frontier_{in}$ is not empty
  - Process all vertices in parallel
    - Update neighbors and add neighbors to $frontier_{out}$
  - Swap $frontier_{in}$ and $frontier_{out}$

Slide source: Courtesy of the 6.106 Lecturers 2022

How do we implement this algorithm in parallel?

Let’s first consider a higher level code

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
    std::set<int> frontier_in, frontier_out;
    frontier_in.insert(source);
    depth[source] = 0;
    int d = 1;

    while (!frontier_in.empty()) {
        #pragma omp parallel for
        for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
            int v = *it;
            for (int neigh : g.neigh(v)) {
                if (depth[neigh] == -1) {
                    depth[neigh] = d;
                    frontier_out.insert(neigh);
                }
            }
        }
        frontier_in = frontier_out;
        frontier_out.clear();
        d++;
    }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
    std::set<int> frontier_in, frontier_out;
    frontier_in.insert(source);
    depth[source] = 0;
    int d = 1;

    while (!frontier_in.empty()) {
        #pragma omp parallel for
        for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
            int v = *it;
            for (int neigh : g.neigh(v)) {
                if (depth[neigh] == -1) {
                    depth[neigh] = d;
                    frontier_out.insert(neigh);
                }
            }
        }
        frontier_in = frontier_out;
        frontier_out.clear();
        d++;
    }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
    std::set<int> frontier_in, frontier_out;
    frontier_in.insert(source);
    depth[source] = 0;
    int d = 1;

    while (!frontier_in.empty()) {
        #pragma omp parallel for
        for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
            int v = *it;
            for (int neigh : g.neigh(v)) {
                if (depth[neigh] == -1) {
                    depth[neigh] = d;
                    frontier_out.insert(neigh);
                }
            }
        }
        frontier_in = frontier_out;
        frontier_out.clear();
        d++;
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
            int v = *it;
            for (int neigh : g.neigh(v)) {
             if (depth[neigh] == -1) {
                depth[neigh] = d;
                frontier_out.insert(neigh);
             }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
               if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
                  depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
                  frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
```

There are race conditions in this code!

What are they?

Breadth-First Search: Parallel

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (depth[neigh] == -1) {
               depth[neigh] = d;
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
}
```

The parallel for loop causes a race condition because a node can be neighbors with many other nodes!

Breadth-First Search: Parallel

How can we fix this race condition?

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
    std::set<int> frontier_in, frontier_out;
    frontier_in.insert(source);
    depth[source] = 0;
    int d = 1;

    while (!frontier_in.empty()) {
        #pragma omp parallel for
        for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
            int v = *it;
            for (int neigh : g.neigh(v)) {
                if (depth[neigh] == -1) {
                    depth[neigh] = d;
                    frontier_out.insert(neigh);
                }
            }
        }
    frontier_in = frontier_out;
```

Breadth-First Search: Parallel

Let's try to use a CAS!

```
void BFS(const Graph& g, int source, std::vector<int>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
            if (CAS(&depth[neigh], -1, d)) {
               frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
   }
}
```

Breadth-First Search: Parallel

Let's try to use a CAS!

```
void BFS(const Graph& g, int source,
std::vector<std::atomic<int>>& depth)
{
   std::set<int> frontier_in, frontier_out;
   frontier_in.insert(source);
   depth[source] = 0;
   int d = 1;

   while (!frontier_in.empty()) {
      #pragma omp parallel for
      for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
         int v = *it;
         for (int neigh : g.neigh(v)) {
    if (depth[neigh].compare_exchange_strong( -1, d))
{
                frontier_out.insert(neigh);
            }
         }
      }
      frontier_in = frontier_out;
      frontier_out.clear();
      d++;
```

Breadth-First Search: Parallel using CSR

```
struct GraphCSR {
    int n;
    std::vector<int> row_ptr;
    std::vector<int> val;
};

void BFS(const GraphCSR& g, int source,
std::vector<std::atomic<int>>& depth) {
    std::set<int> frontier_in, frontier_out;

    frontier_in.insert(source);
    depth[source].store(0,
std::memory_order_relaxed);

    int d = 1;
```

```
    while (!frontier_in.empty()) {
     #pragma omp parallel for
     for (auto it = frontier_in.begin(); it != frontier_in.end(); ++it) {
        int v = *it;
for (int idx = g.row_ptr[v]; idx < g.row_ptr[v+1]; idx++) {
            int neigh = g.val[idx];

            if (depth[neigh].compare_exchange_strong(-1, d)) {
               frontier_out.insert(neigh);
            }
        }
     }

     frontier_in = frontier_out;
     frontier_out.clear();
     d++;
   }
}
```

Page Rank

The algorithm that started Google!

Page Rank

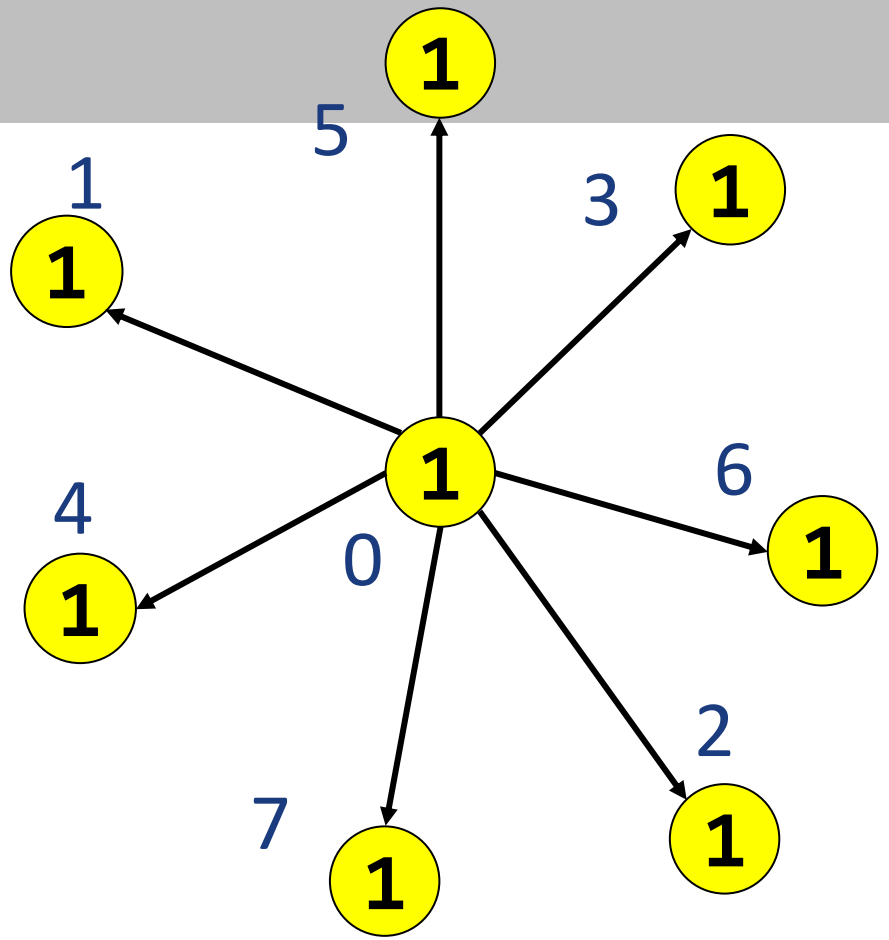
- Problem: Assign a score to each vertex based on the scores of neighbors

"The Anatomy of a Large-Scale Hypertextual Web Search Engine"
by Sergey Brin and Lawrence Page
in Computer Networks, vol. 30 (1998)

- Measures importance of web pages based on hyperlinks and scores of links
- Each node has score indicating importance
- Higher score means many high-scoring nodes link to it
- Create a graph where **vertices are pages** and **edges are links**

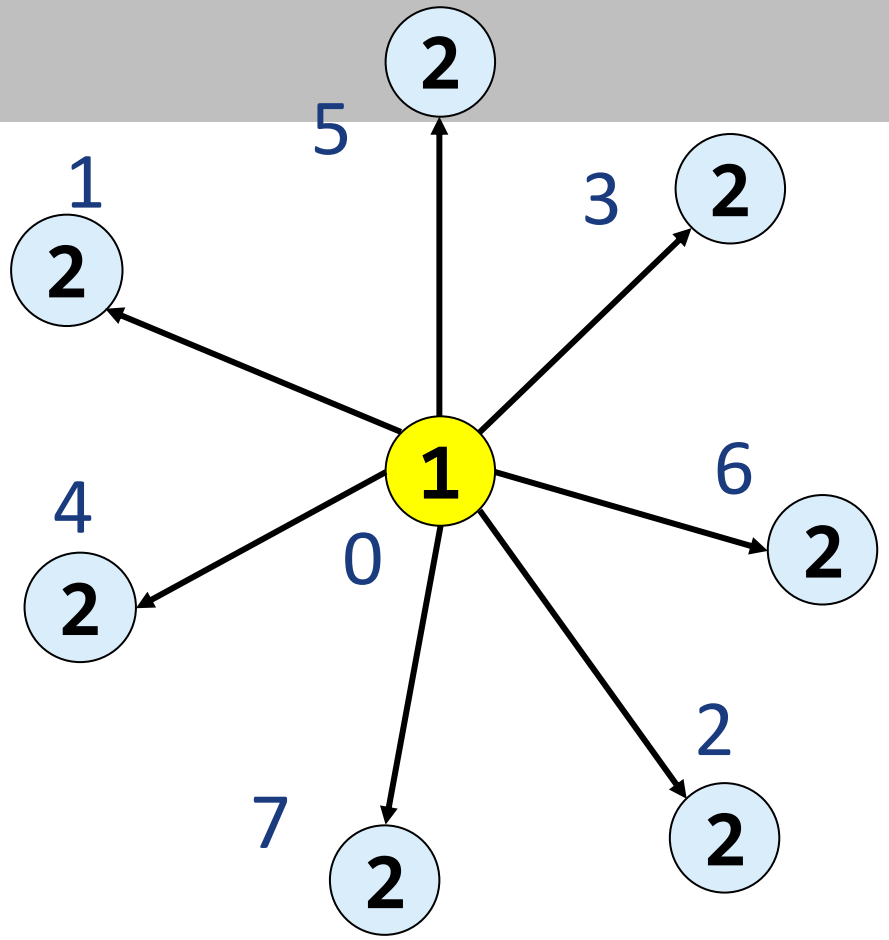
Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Algorithm Example



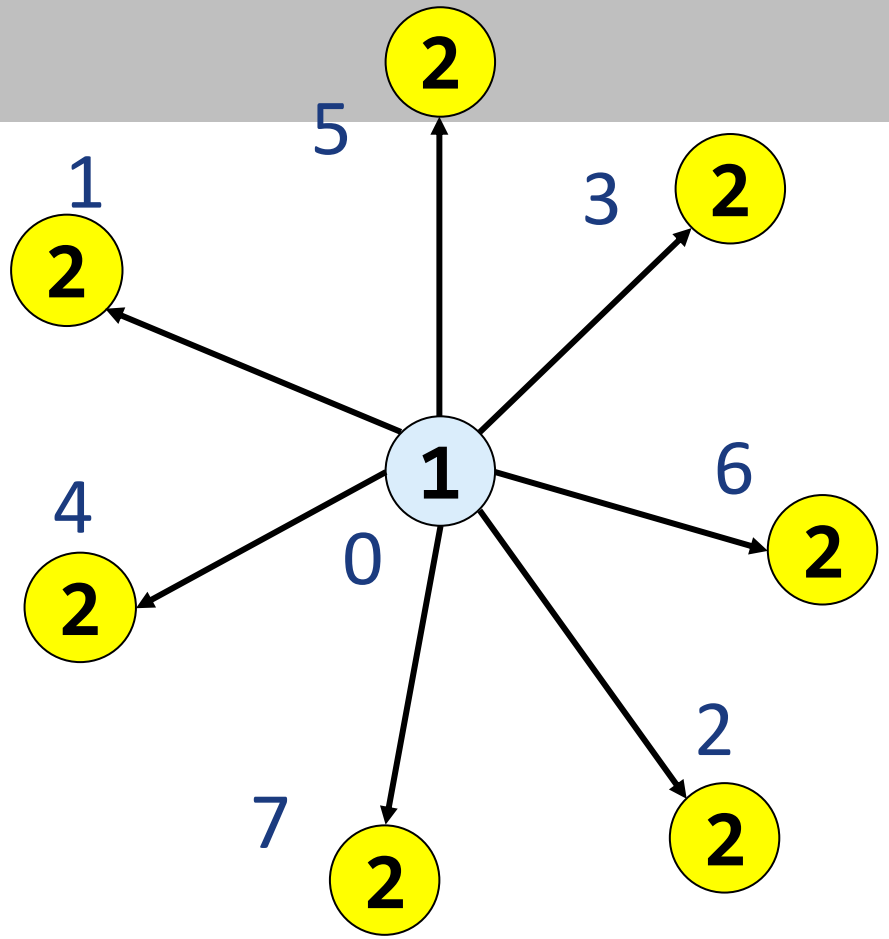
- Initialize all scores to 1

Page Rank Algorithm Example



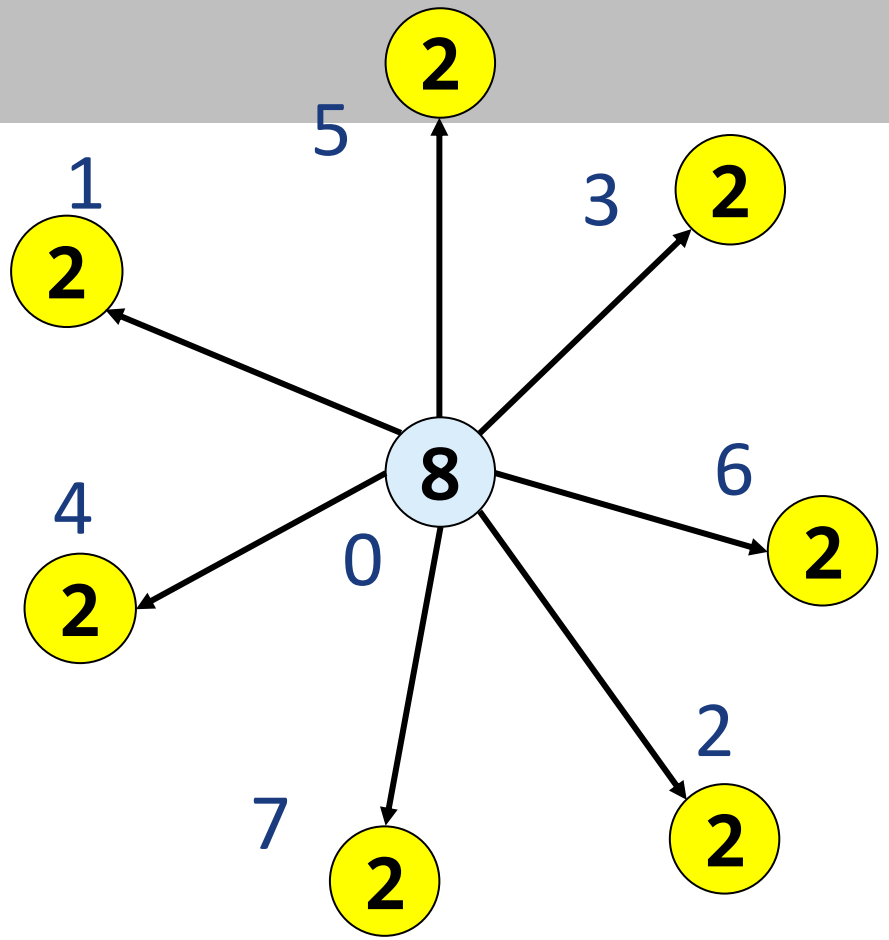
- Initialize all scores to 1
- Send your score to all your neighbors and neighbors calculate new score by adding your score

Page Rank Algorithm Example



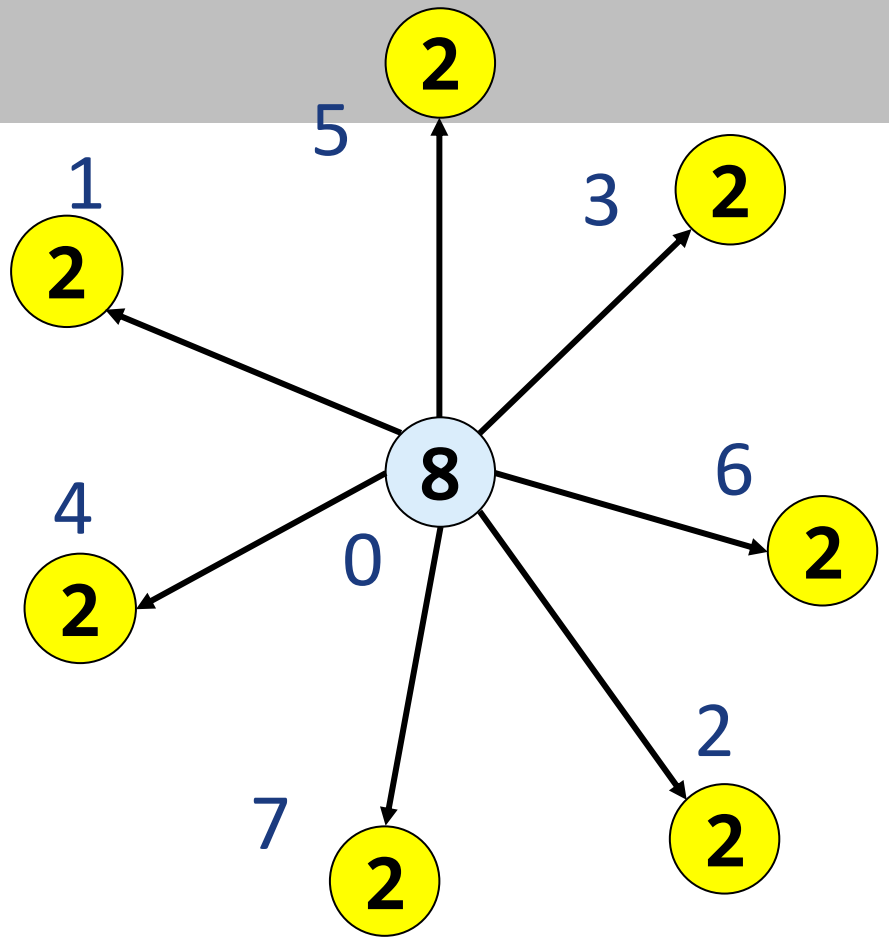
- Initialize all scores to 1
- Send your score to all your neighbors and neighbors calculate new score by adding your score

Page Rank Algorithm Example



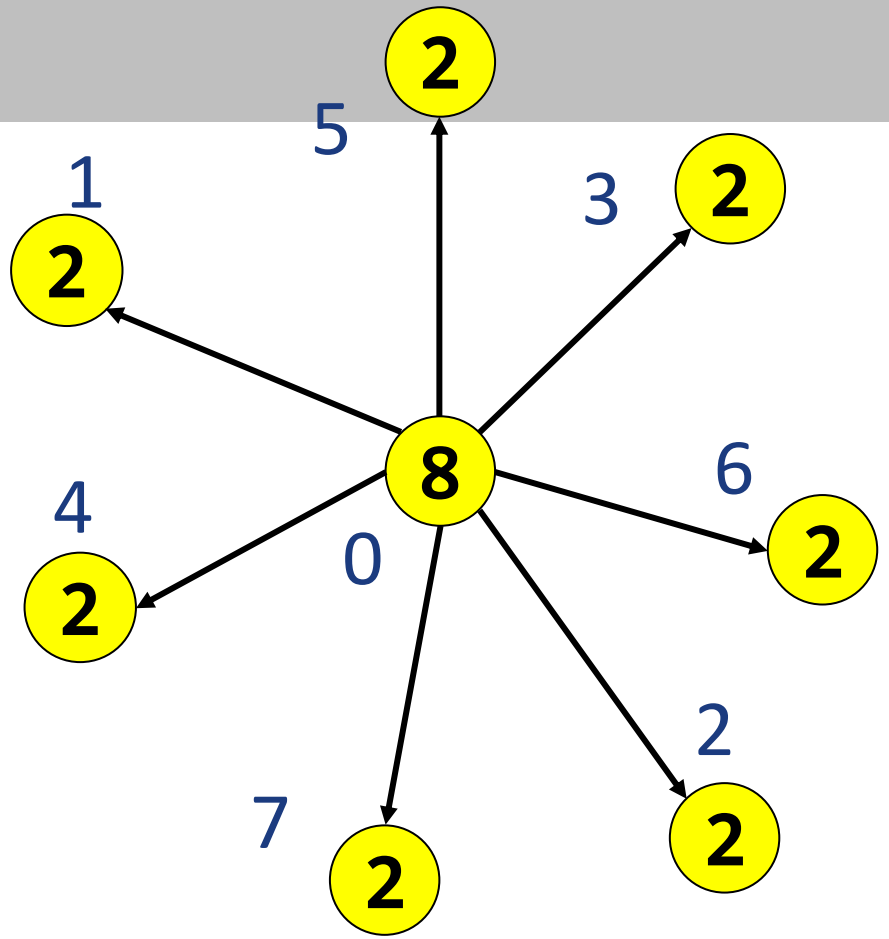
- Initialize all scores to 1
- Send your score to all your neighbors and neighbors calculate new score by adding your score

Page Rank Algorithm Example



- Initialize all scores to 1
- Send your score to all your neighbors and neighbors calculate new score by adding your score
- Proceed for multiple rounds

Page Rank Algorithm Example



- Initialize all scores to 1
- Send your score to all your neighbors and neighbors calculate new score by adding your score
- Proceed for multiple rounds

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
    #pragma omp parallel for
    for (auto v : g) {
        for (auto neigh : g.neigh(v)) {
            new_ranks[neigh] += old_ranks[v];
        }
    }
}
```

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
   #pragma omp parallel for
   for (auto v : g) {
      for (auto neigh : g.neigh(v)) {
         new_ranks[neigh] += old_ranks[v];
      }
   }
}
```

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
   #pragma omp parallel for
   for (auto v : g) {
      for (auto neigh : g.neigh(v)) {
         new_ranks[neigh] += old_ranks[v];
      }
   }
}
```

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
    #pragma omp parallel for
    for (auto v : g) {
        for (auto neigh : g.neigh(v)) {
            new_ranks[neigh] += old_ranks[v];
        }
    }
}
```

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
    #pragma omp parallel for
    for (auto v : g) {
        for (auto neigh : g.neigh(v)) {
            new_ranks[neigh] += old_ranks[v];
        }
    }
}
```

Where is the race condition?

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
   #pragma omp parallel for
   for (auto v : g) {
      for (auto neigh : g.neigh(v)) {
         new_ranks[neigh] += old_ranks[v];
      }
   }
}
```

How do we fix this race condition in this code?

Page Rank Parallel Implementation

```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
   #pragma omp parallel for
   for (auto v : g) {
      for (auto neigh : g.neigh(v)) {
          #pragma omp atomic
          new_ranks[neigh] += old_ranks[v];
      }
   }
}
```

Use an atomic add instead!

Page Rank Parallel Implementation

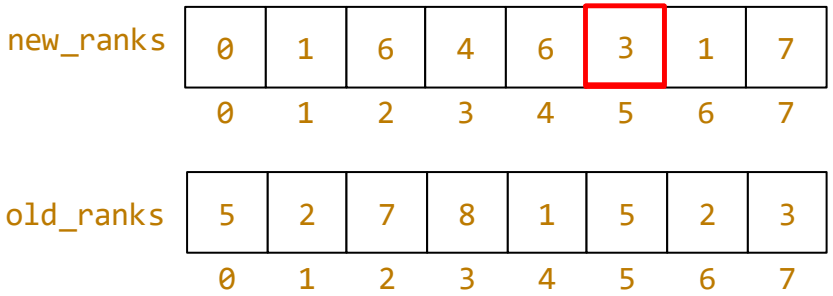
```
void PageRank(Graph g, float old_ranks[], float new_ranks[]) {
   #pragma omp parallel for
   for (auto v : g) {
      for (auto neigh : g.neigh(v)) {
          #pragma omp atomic
          new_ranks[neigh] += old_ranks[v];
      }
```

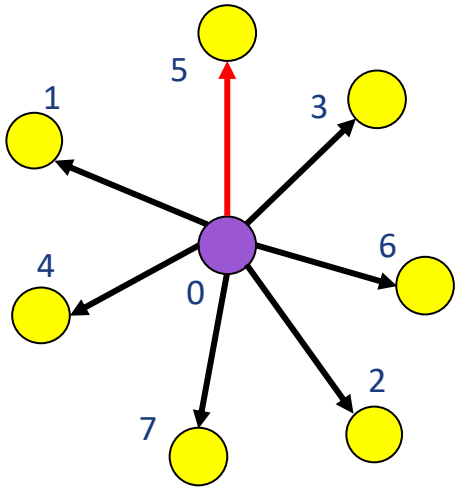
What are things that affect performance poorly in this algorithm?

Use an atomic add instead!

Page Rank Parallel Implementation

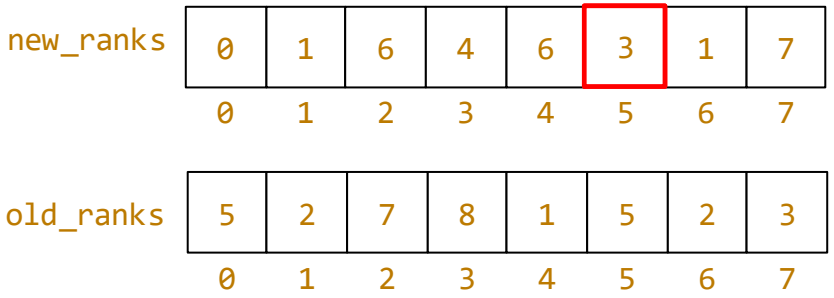
- PageRank has poor cache performance!

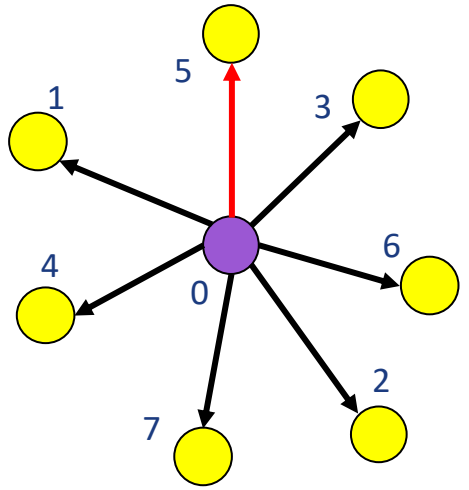




Slide source: Courtesy of the 6.106 Lecturers 2022

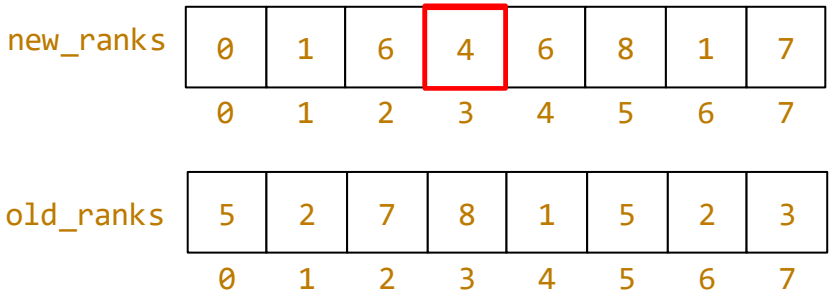
Page Rank Parallel Implementation

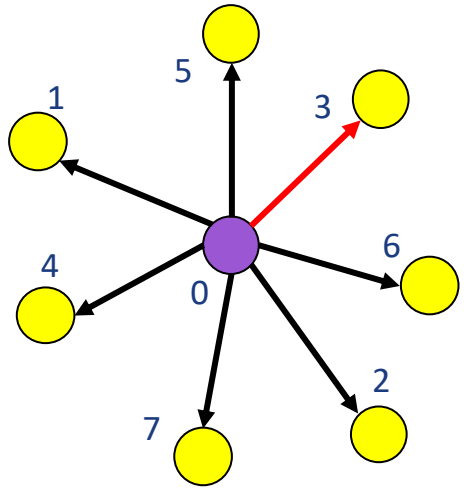




Slide source: Courtesy of the 6.106 Lecturers 2022

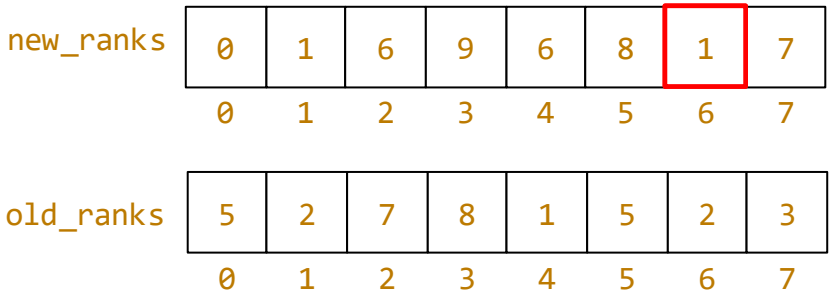
Page Rank Parallel Implementation

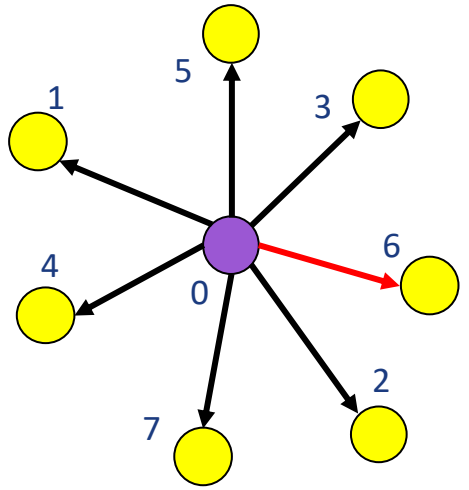




Slide source: Courtesy of the 6.106 Lecturers 2022

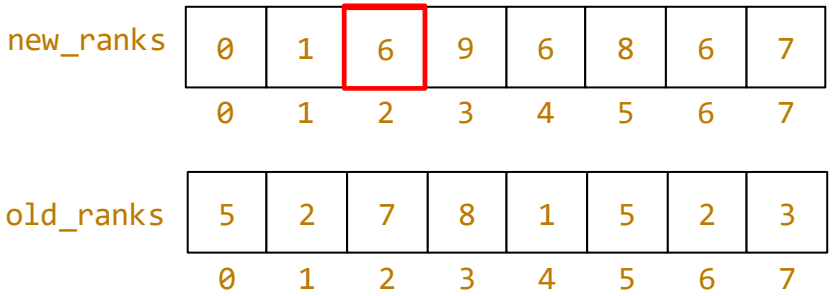
Page Rank Parallel Implementation

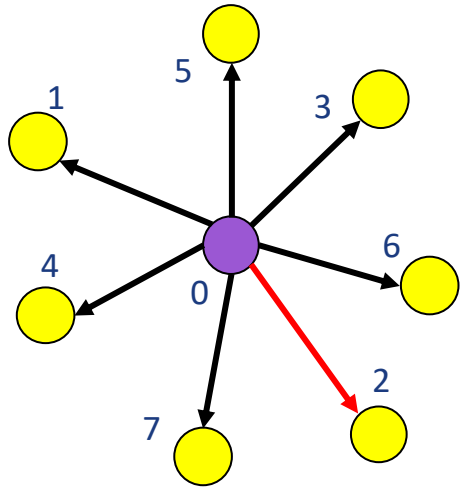




Slide source: Courtesy of the 6.106 Lecturers 2022

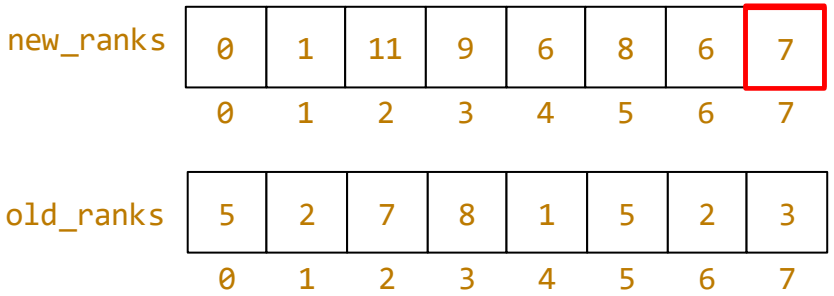
Page Rank Parallel Implementation

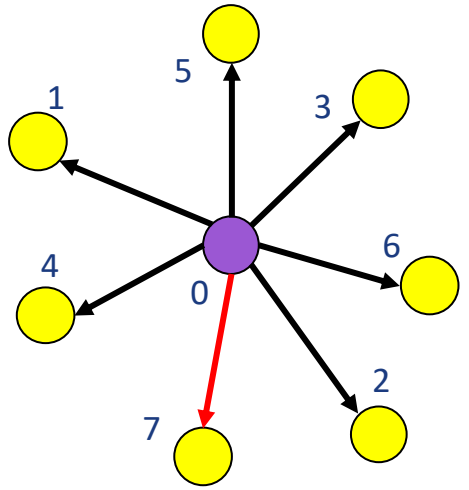




Slide source: Courtesy of the 6.106 Lecturers 2022

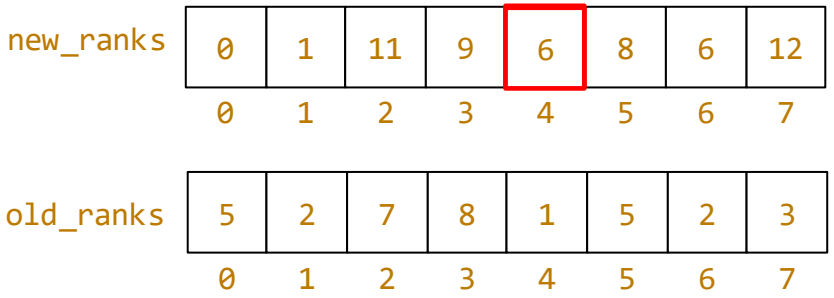
Page Rank Parallel Implementation

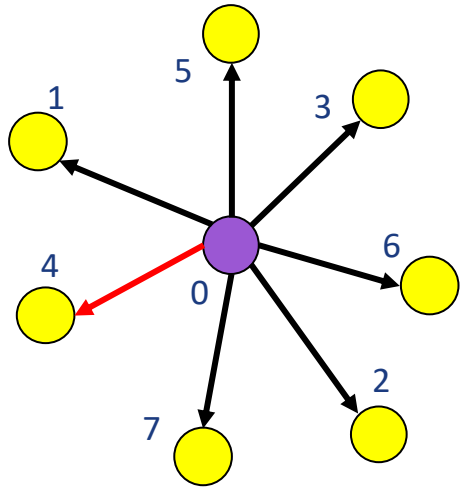




Slide source: Courtesy of the 6.106 Lecturers 2022

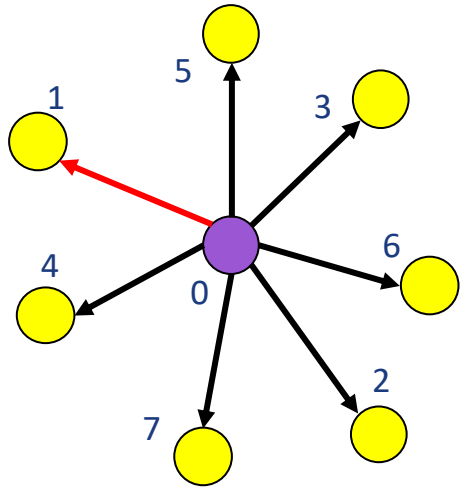
Page Rank Parallel Implementation





Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation



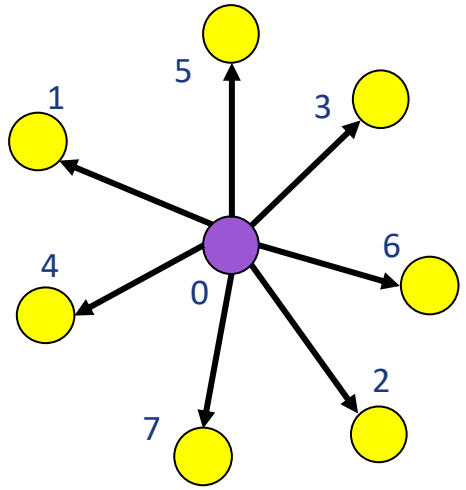
Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Every write is a random access in the new_ranks array!

new_ranks	0	6	11	9	11	8	6	12
	0	1	2	3	4	5	6	7

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



Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- PageRank has poor cache performance

Page Rank Parallel Implementation

- PageRank has poor cache performance
- Every write is a random access in the new_ranks array

Page Rank Parallel Implementation

- PageRank has poor cache performance
- Every write is a random access in the new_ranks array
- How do we make this less of a random access algorithm?

Page Rank Parallel Implementation

- PageRank has poor cache performance
- Every write is a random access in the new_ranks array
- How do we make this less of a random access algorithm?
- Partition the edges and store them in contiguous arrays

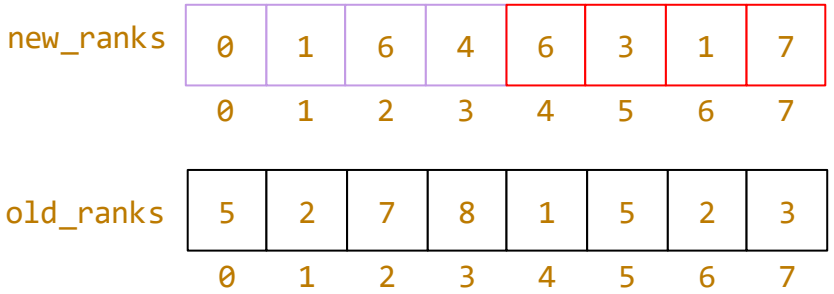
Page Rank Parallel Implementation

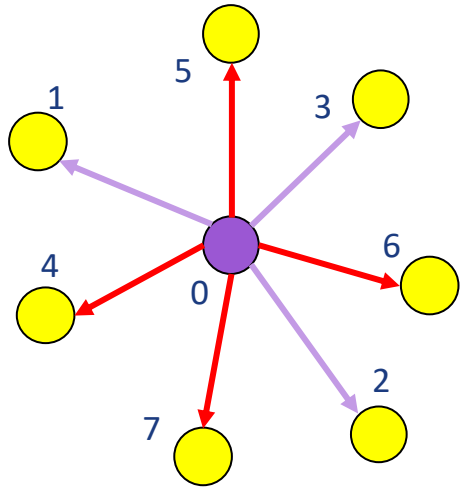
- PageRank has poor cache performance
- Every write is a random access in the new_ranks array
- How do we make this less of a random access algorithm?
- Partition the edges and store them in contiguous arrays
- Color each contiguous array in parallel

Page Rank Parallel Implementation

- PageRank has poor cache performance
- Every write is a random access in the new_ranks array
- How do we make this less of a random access algorithm?
- Partition the edges and store them in contiguous arrays
- Color each contiguous array in parallel
- Edges are colored based on destination partition

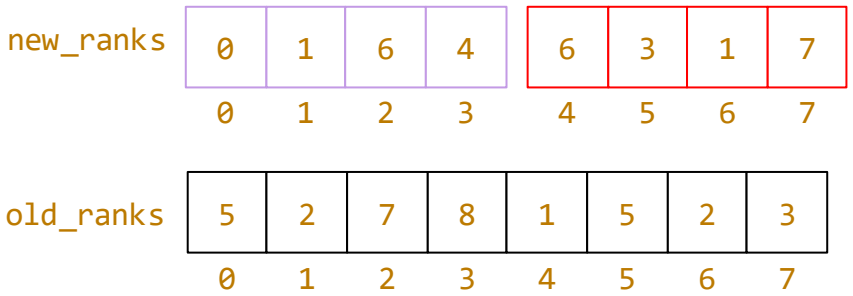
Page Rank Parallel Implementation

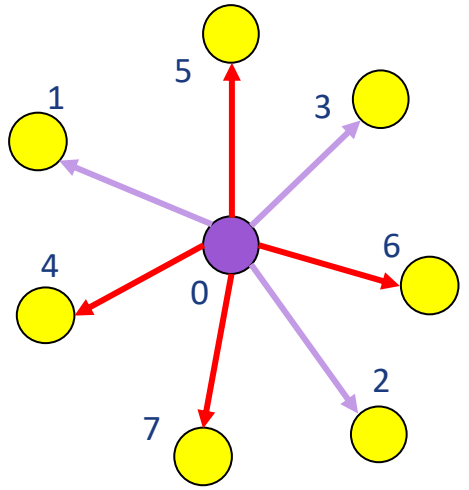




Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

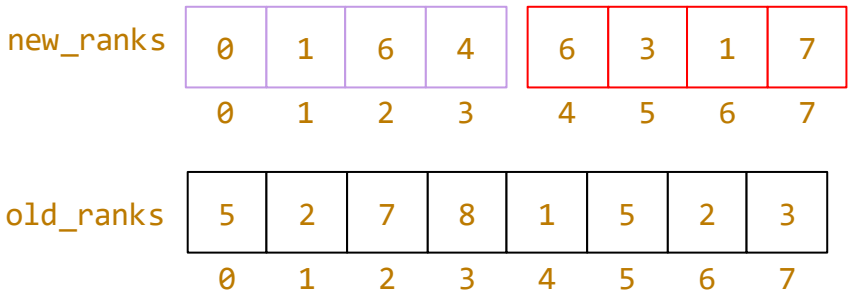


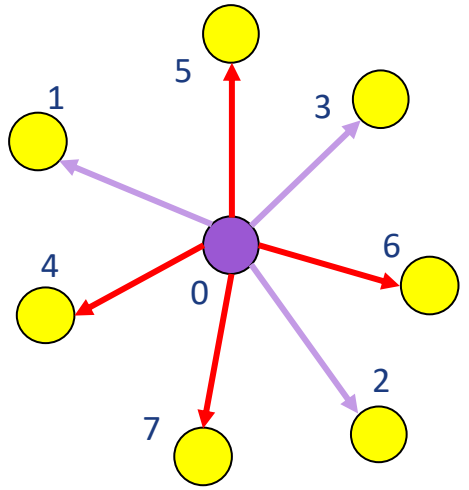


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

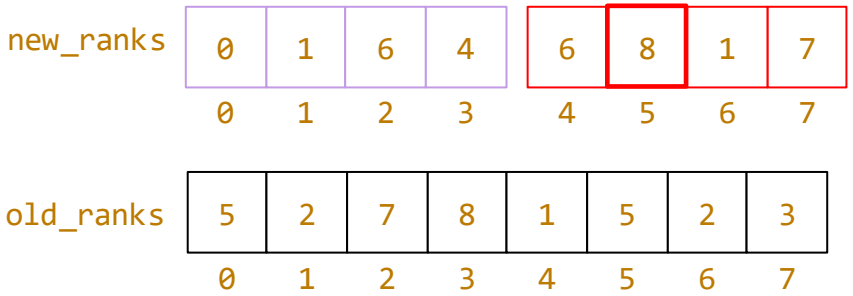


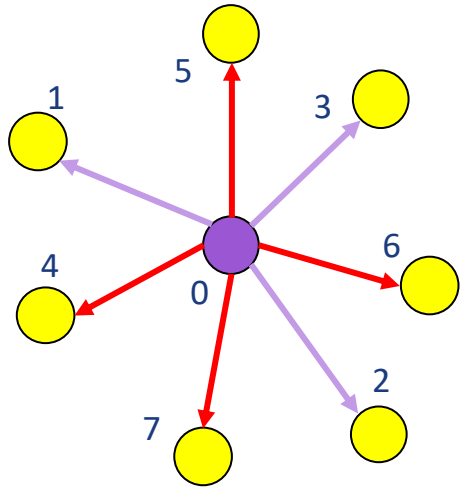


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

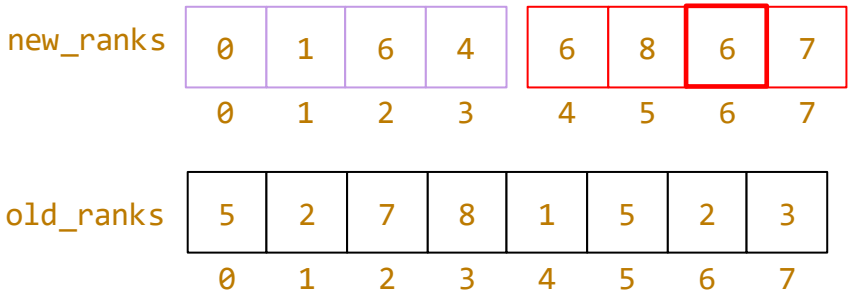


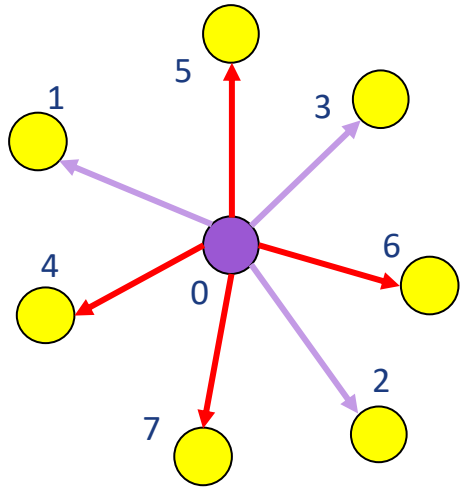


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

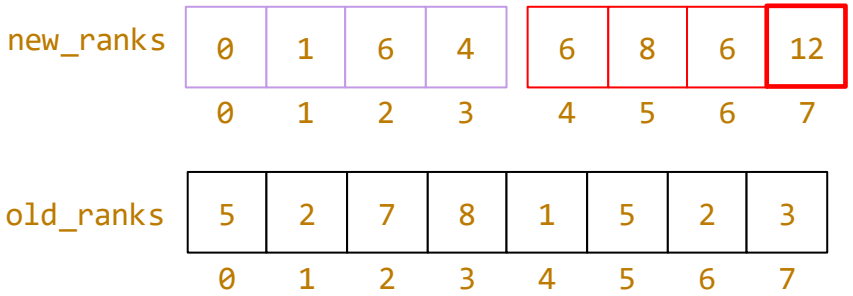


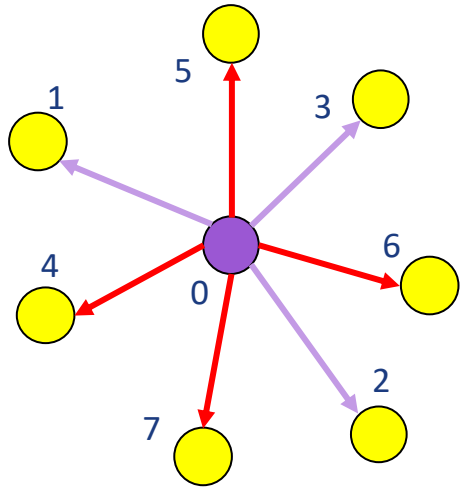


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

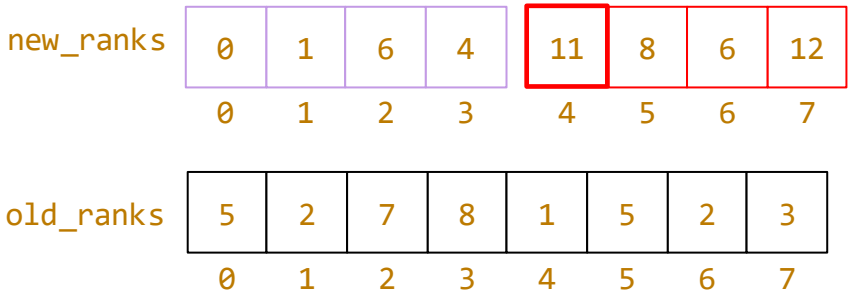


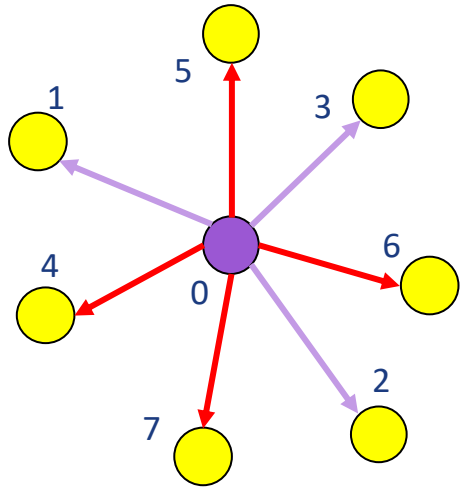


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

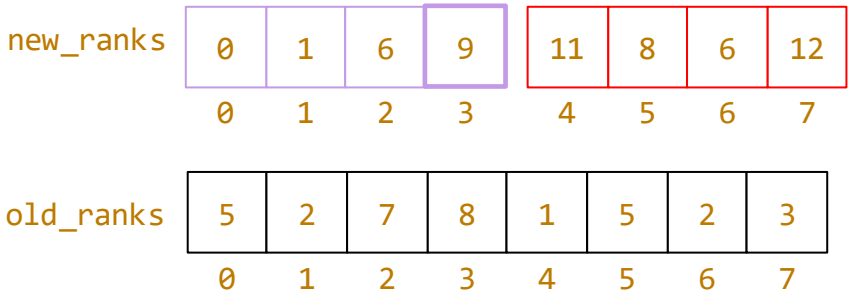


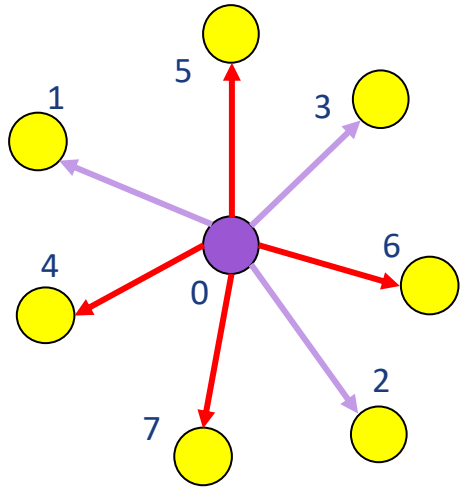


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

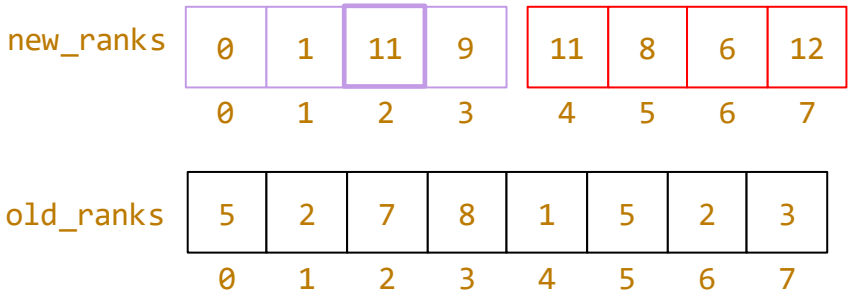


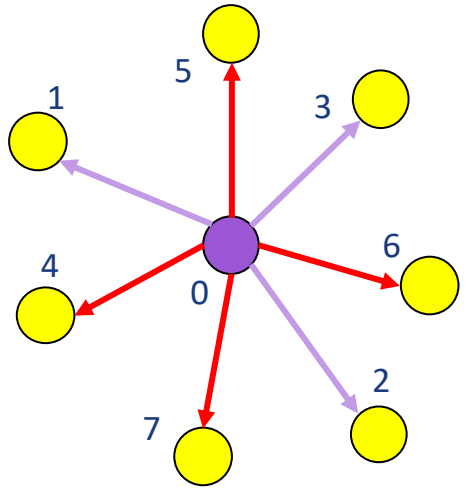


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

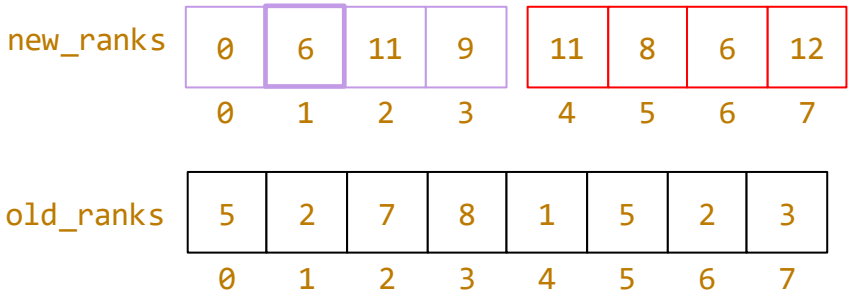


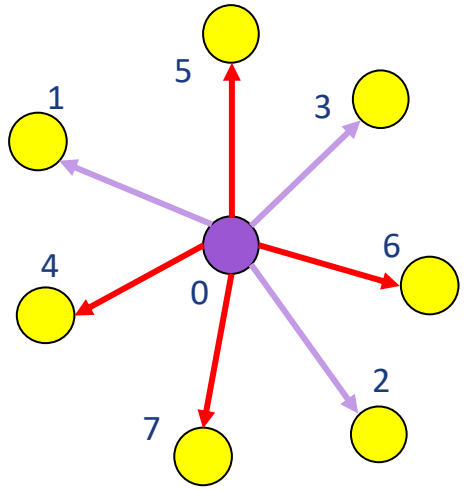


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other

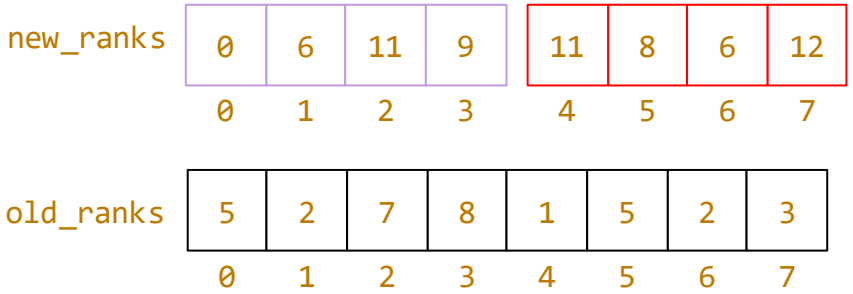


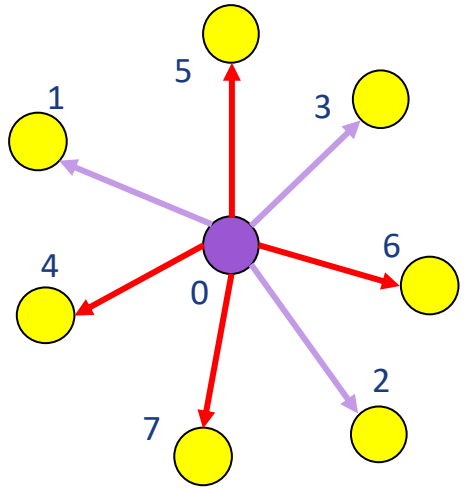


Slide source: Courtesy of the 6.106 Lecturers 2022

Page Rank Parallel Implementation

- Iterate through each partition one after the other





- All random accesses fit within the same cache

Page Rank More Details

- Blocking for cache requires pre-partitioning the graph

- Convert the data layout from **CSR** to **BCSR** (Blocked CSR)
  - Stores non-zero elements as small dense blocks
  - row_ptr refers to a row of blocks instead of a row
  - col stores columns of the blocks
  - val stores dense array of values
- Since Pagerank typically runs 100s of iterations, cost of partitioning is amortized across all iterations (one upfront cost)

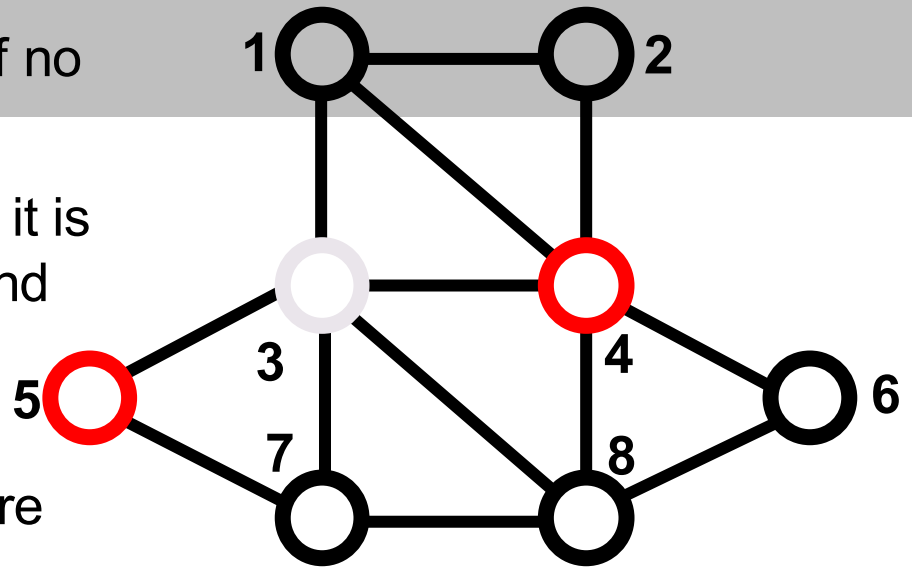
Slide source: Courtesy of the 6.106 Lecturers 2022

Parallel Maximal Independent Set

Another example

Maximal Independent Set

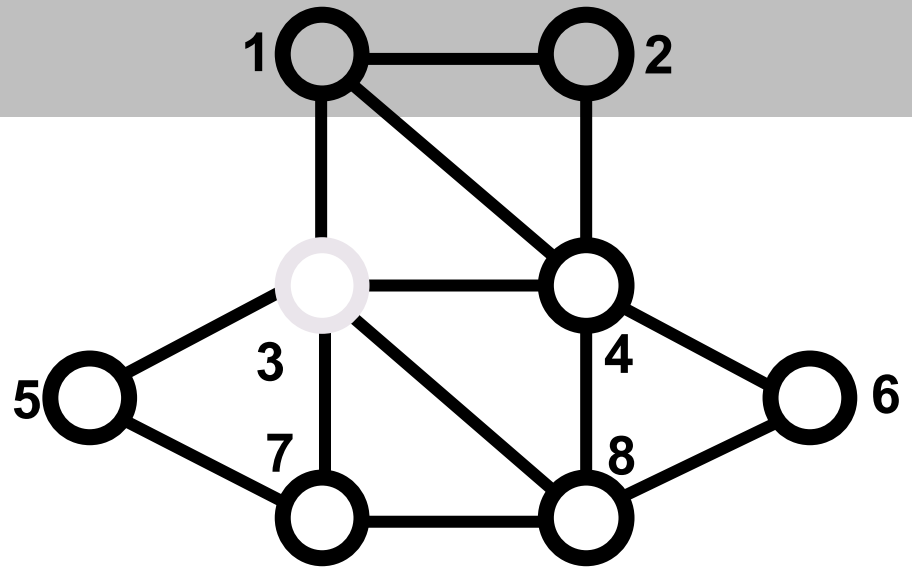
- Graph with vertices V = {1,2,…,n}
- A set S of vertices is independent if no two vertices in S are neighbors.
- An independent set S is maximal if it is impossible to add another vertex and stay independent
- An independent set S is maximum if no other independent set has more vertices
- Finding a *maximum* independent set is intractably difficult (NP-hard)
- Finding a *maximal* independent set is easy, at least on one processor.



The set of red vertices
S = {4, 5} is *independent*
and is *maximal*
but not *maximum*

Sequential Maximal Independent Set

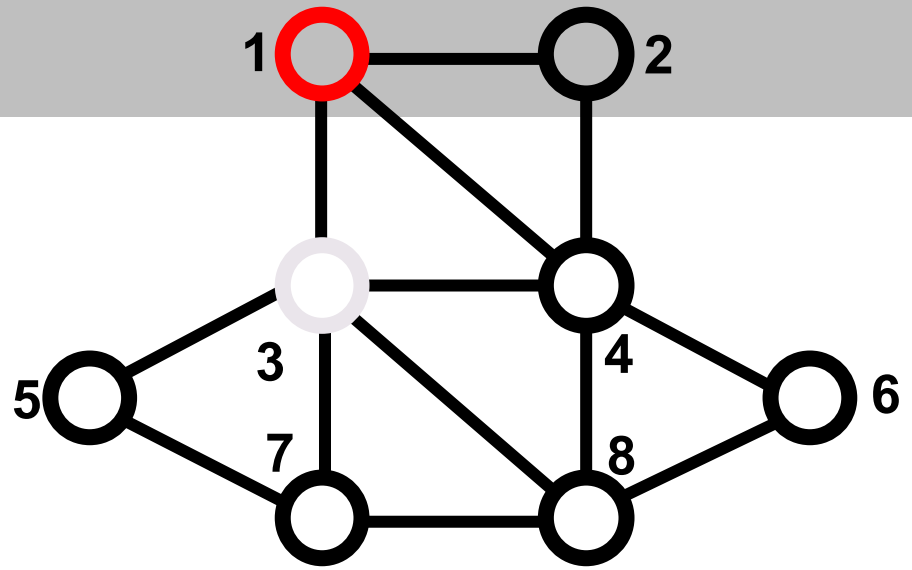
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { }

Sequential Maximal Independent Set

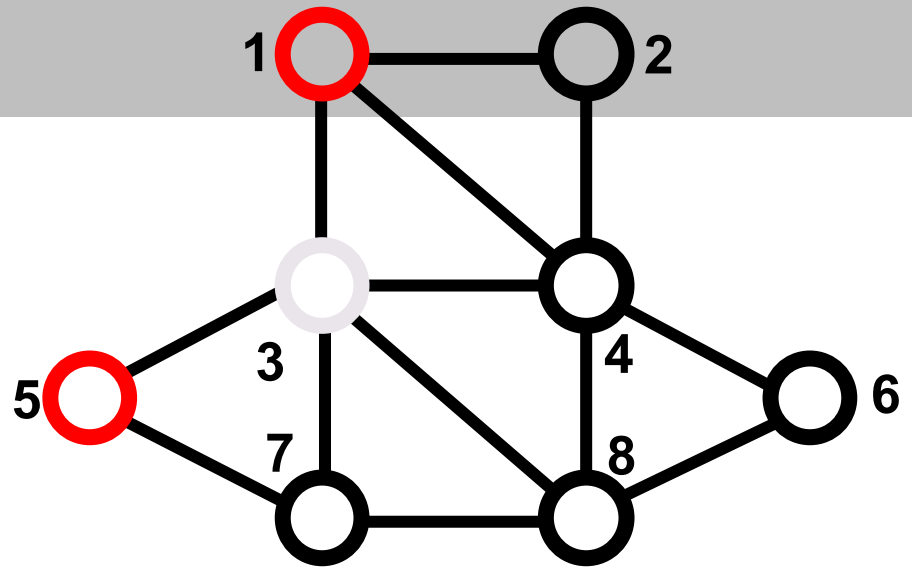
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1 }

Sequential Maximal Independent Set

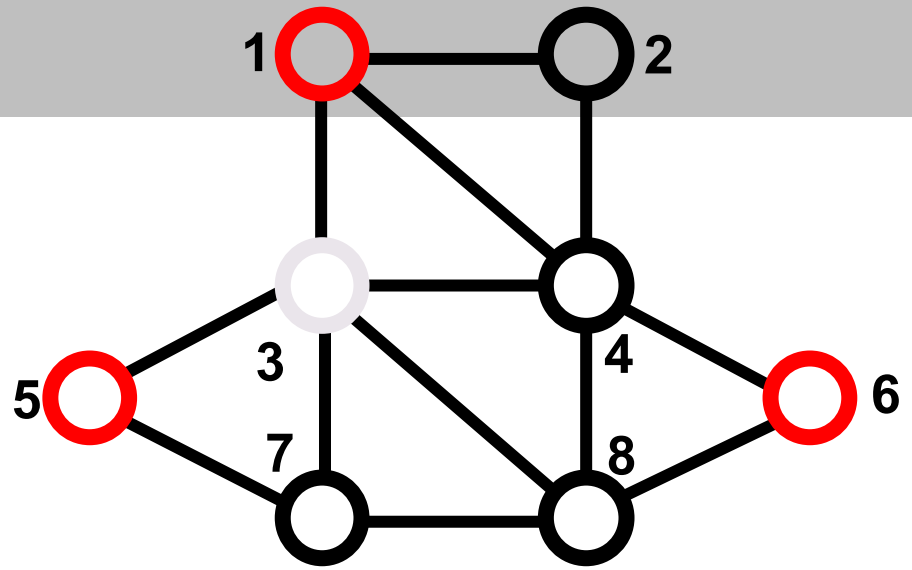
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1, 5 }

Sequential Maximal Independent Set

```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```

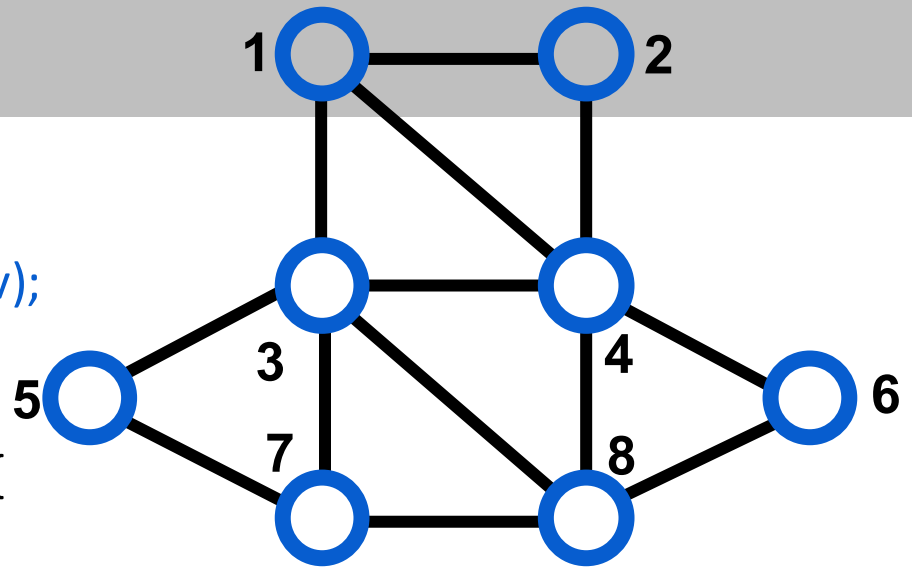


S = { 1, 5, 6 }

work **O(n), but** ***span*** **O(n)**

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



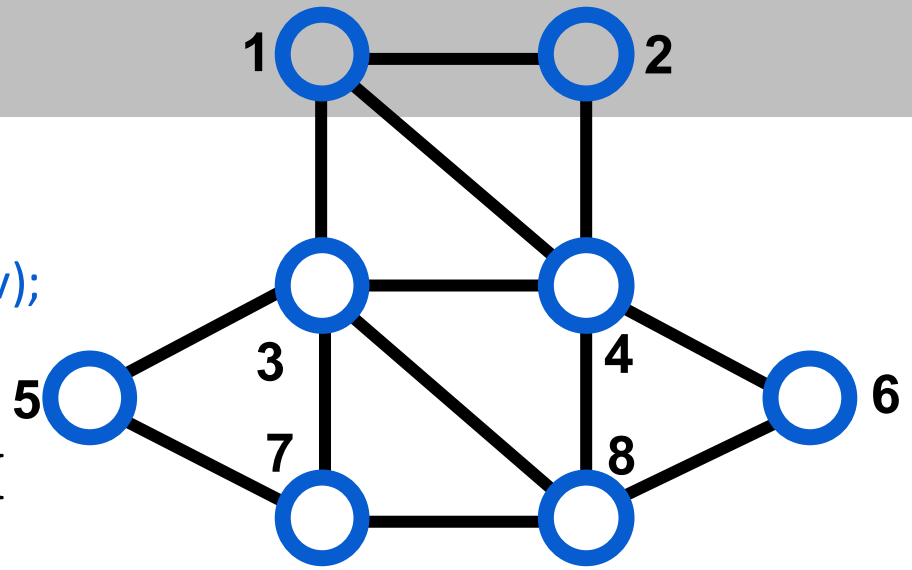
S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

M. Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". *SIAM Journal on Computing* **15** (4): 1036–1053, 1986

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

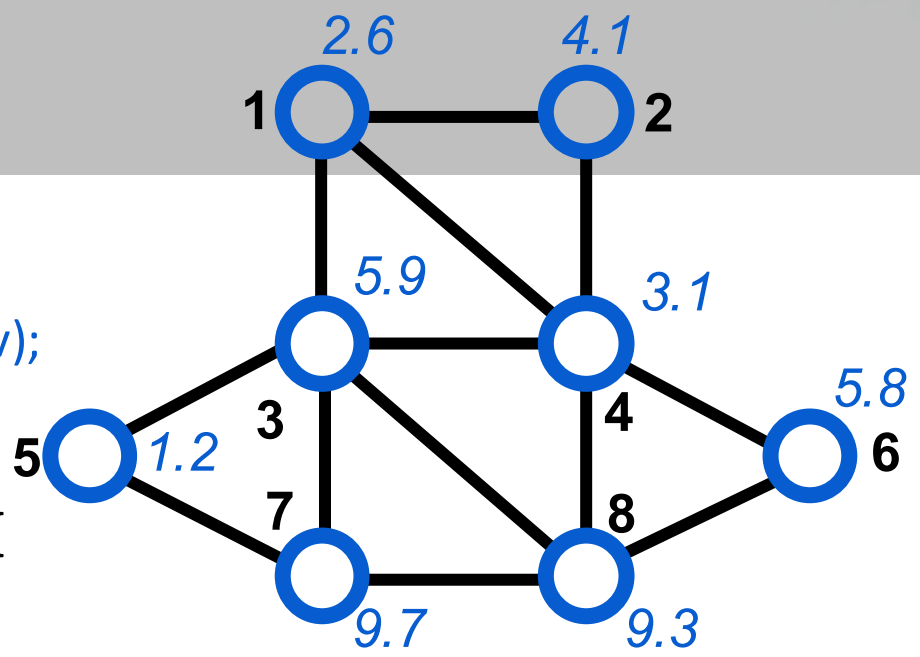


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

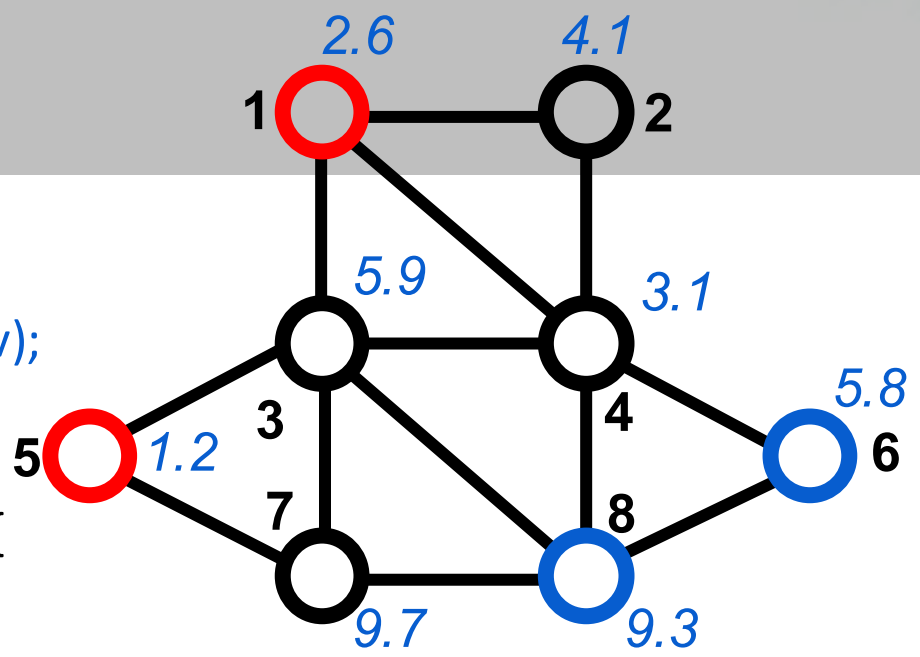


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

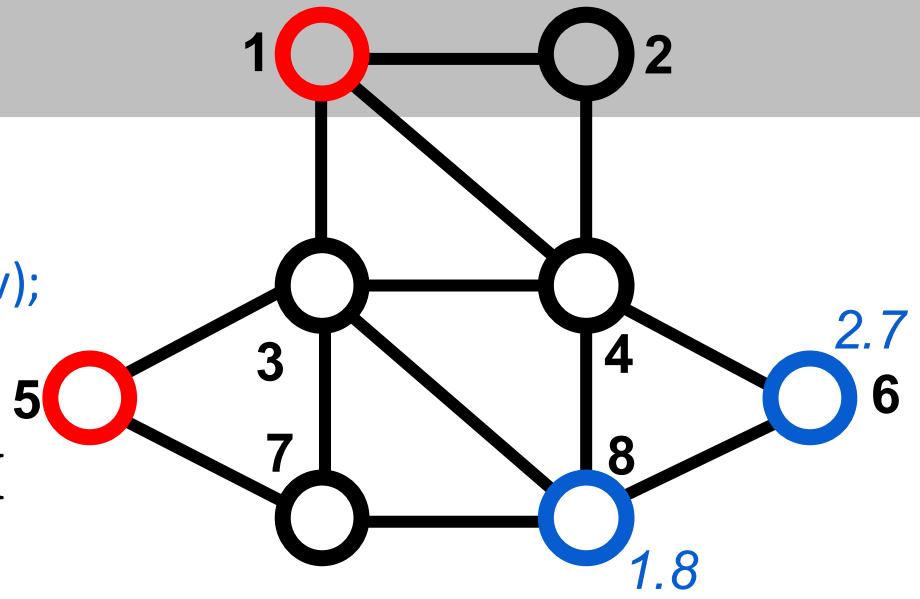


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

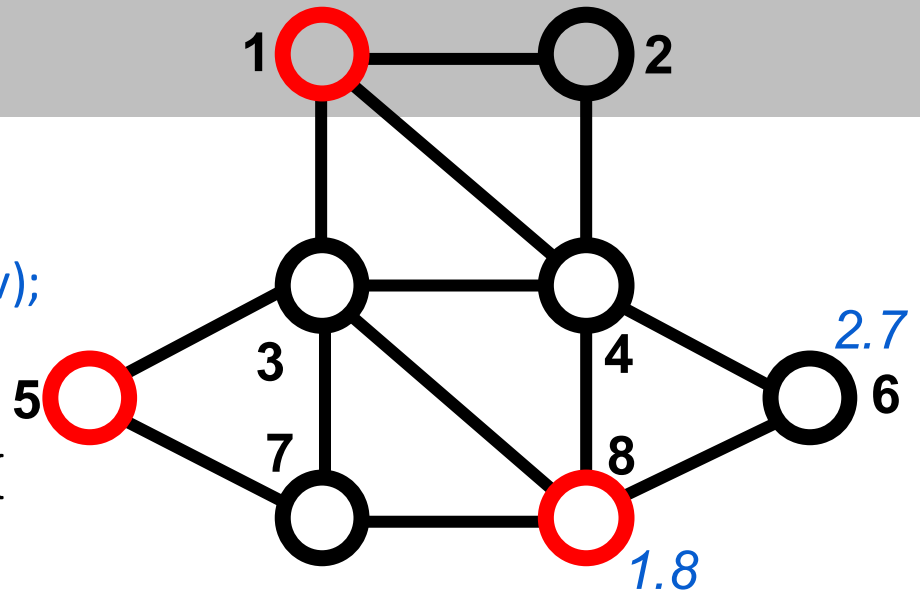


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

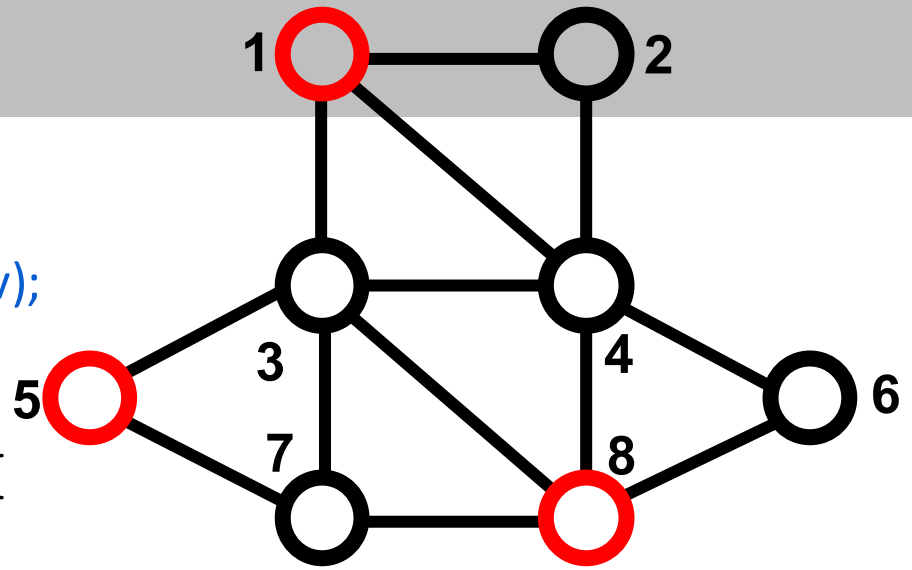


S = { 1, 5, 8 }

C = { }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



Theorem: This algorithm "very probably" finishes within O(log n) rounds.

work **O(n log n), but** ***span*** **O(log n)**

Lecture 13 Quiz