# Randomized Algorithms

# Last time …

- Network topologies
- Intro to MPI
- Matrix-matrix  multiplication

# Today …

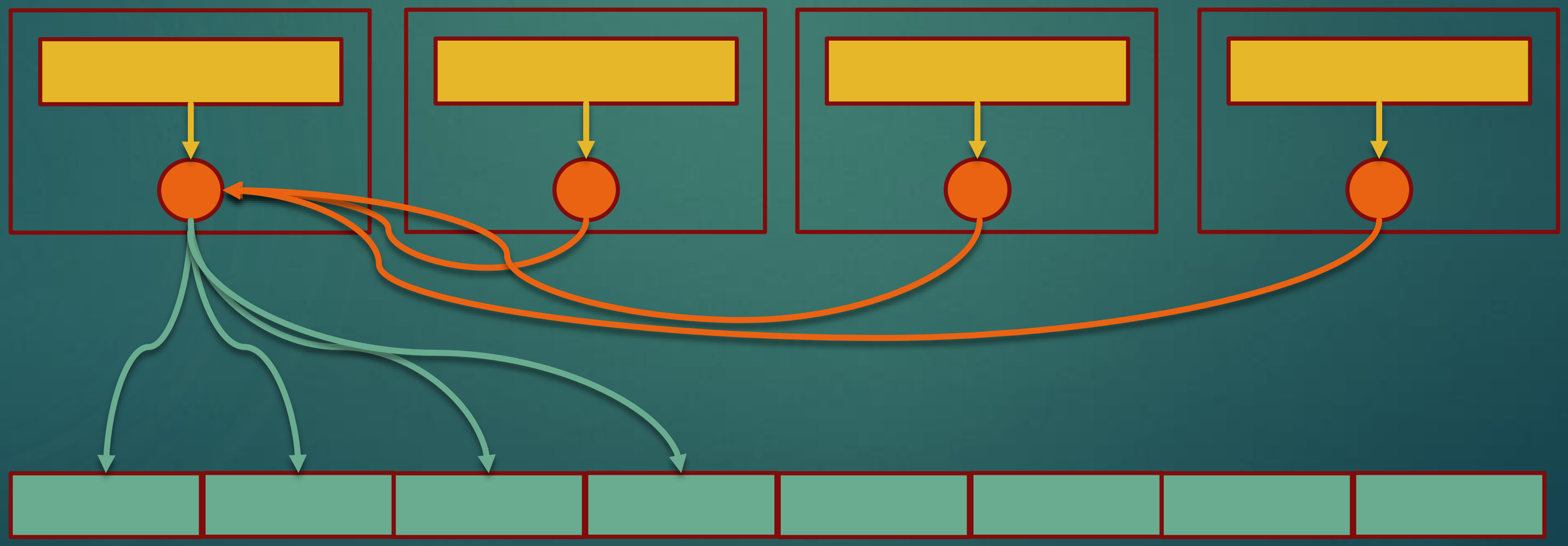
- MPI I/O
- Randomized Algorithms
  - Parallel $k$-Select
  - Graph coloring
- Assignment 2

# Parallel I/O

- Goal of Parallel I/O is to use parallelism to increase bandwidth
- Parallel I/O can be hard to coordinate and optimize if working directly at the level of Lustre(DFS) API or POSIX I/O Interface
- Therefore, specialists implement a number of intermediate layers for coordination of data access and mapping from application layer to I/O layer
- Hence, application developers only have to deal with a high-level interface built on top of a software stack that in turn sits on top of the underlying hardware

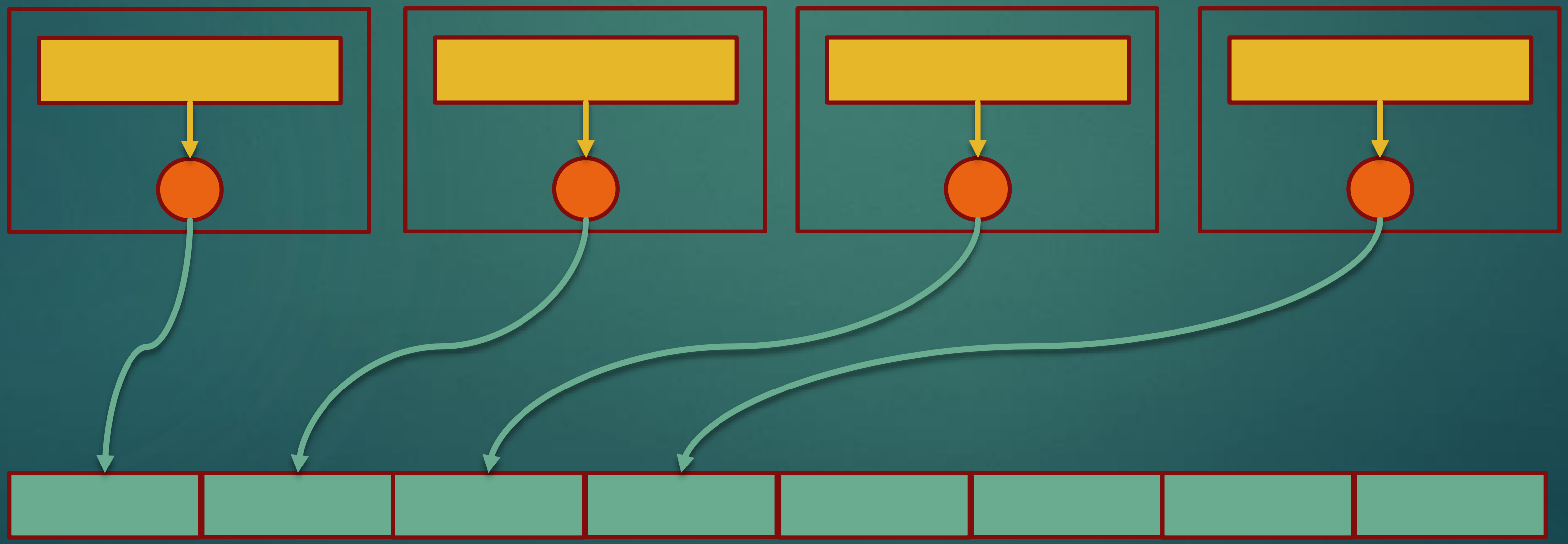
# Typical Pattern: Parallel programs doing sequential I/O

- All processes send data to master process, and then the process designated as master writes the collected data to the file
- This sequential nature of I/O can limit performance and scalability of many applications



# Desired Pattern: Parallel Programs doing Parallel I/O

- Multiple processes participating in reading data from or writing data to a common file in parallel
- This strategy improves performance and provides a single file for storage and transfer purposes



**Simple Solution?**

# MPI for Parallel I/O

- Reading and writing in parallel is like receiving and sending messages
- Hence, an MPI-like machinery is a good setting for Parallel I/O (think MPI communicators and MPI datatypes)
- MPI-I/O featured in MPI-2 which was released in 1997, and it interoperates with the file system to enhance I/O performance for distributed memory applications
- Given N number of processes, each process participates in reading or writing a portion of a common file

# Using MPI I/O

- `MPI_File_open, MPI_File_close`
- There are three ways of positioning where the read or write takes place for each process:
  - Use individual file pointers (e.g., `MPI_File_seek/MPI_File_read`)
  - Calculate byte offsets (e.g., `MPI_File_read_at`)
  - Access a shared file pointer (e.g., `MPI_File_seek_shared, MPI_File_read_shared`)
- `Collective I/O`
  - forces all processes in the communicator to read/write data simultaneously and to wait for each other
  - `MPI_File_read_(at)_all, MPI_File_write_(at)_all,`

# Randomized Algorithms

# Recall …

- Two main strategies for parallelization
  - Divide & Conquer
  - **Randomization**

Ensure that processors can make local decisions which,
with *high probability*,
add up to good global decisions

**Sampling → quicksort, samplesort**

# Randomization

- Sampling
- Symmetry breaking
  - Independent sets → today
- Load balancing

# Parallel $k$-Select

- Given an array $A$ with $n$ elements,
- Select$(A, k)$ returns the $k^{th}$ largest element in $A$
- Select$(A, n/2)$, Select$(A, \frac{n+1}{2})$ → median selection
- Select$(A, 1)$ → minimum
- Select$(A, n)$ → maximum

- Note that for sample sort, we can perform $(p-1)$ $k$-selects simultaneously to obtain the $p-1$ splitters

$$\text{Select}\left(A, \frac{n}{p}, \frac{2n}{p}, \dots, \frac{(p-1)n}{p}\right)$$

# Parallel $k$-Select

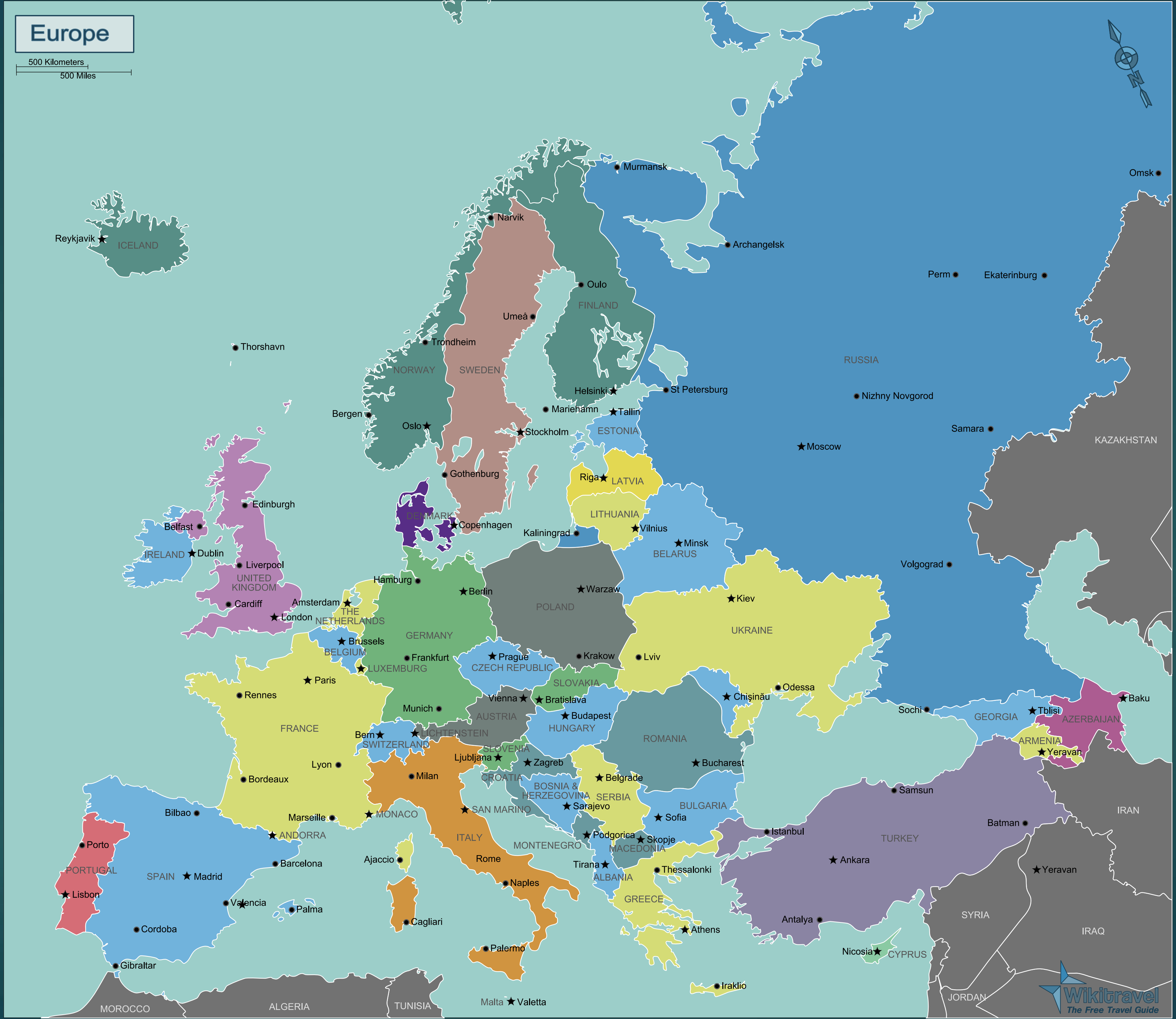
- Select Splitter(s) randomly → $(k_1, k_2)$
- Estimate ranks for the splitters $(R_1, R_2)$ and partition $A$ into three sets
  - Sequential ranking (count) on each process → $\mathcal{O}\left(\frac{n}{p}\right)$
  - Reduction to obtain global ranking → $\mathcal{O}(\log p)$
- Recurse on the appropriate set
  - depending on whether $k \in [0, R_1), [R_1, R_2), \text{or } [R_2, n)$
  - Recursion on a strictly smaller set → will terminate
- Tradeoff between number of splitters at each iteration and number of iterations to converge
  - Choosing $p - 1$ splitters efficient for samplesort
  - What to do for quicksort?

# Graph Algorithms

- Will be covered in detail later …
- Graph $\mathcal{G} = (V, E)$, vertices and edges
- Matrices → Graphs
  - Adjacency graph of a matrix $A$
  - Edge $(i, j)$ exists iff $A_{ij} \neq 0$
    - Edge weight, $W_{ij}$, can be the $A_{ij}$ value
- Graphs → Matrices
  - Adjacency matrix of a weighted graph
    - Default weight 1, vertex value is in-degree
  - Symmetric → undirected graphs
  - Unsymmetric → directed graphs

# Graph Algorithms

- Graph partitioning
  - NP Hard problem
  - We will cover in detail
  - Coloring
- Graph Laplacian & Eigenproblem
- Breadth First Search (BFS)
- Depth First Search (DFS)
- Connected components
  - Spanning Trees




CS 5965/6965 - Big Data Systems - Fall 2014

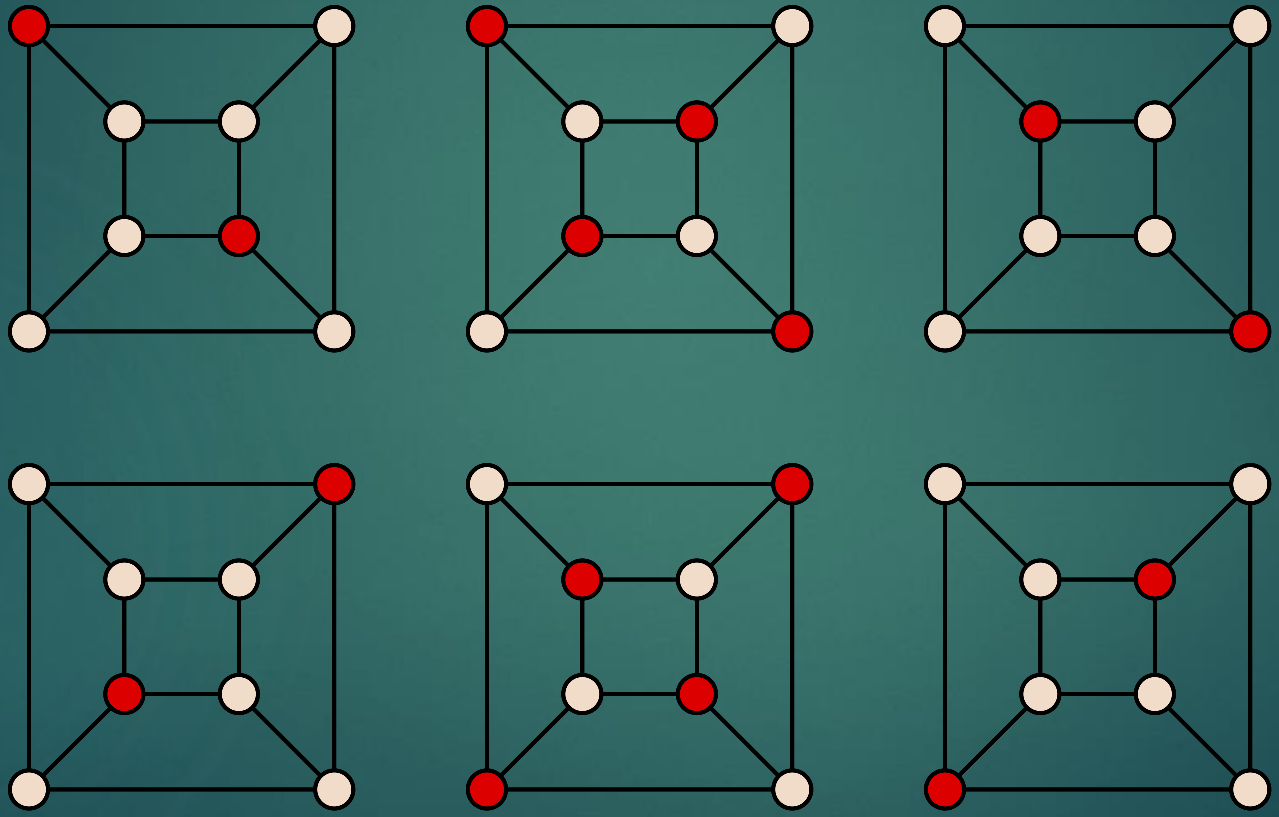
# Sequential Greedy Algorithm

- $n = |V|$
- Choose a random permutation $p(1), \ldots, p(n)$ of numbers $1, \ldots, n$
- $U \leftarrow V$
- for $i \leftarrow 1$ to $n$
  - $v \leftarrow p(i)$
  - $S \leftarrow \{\text{colors of all colored neighbors of } v\}$
  - $c(v) \leftarrow \text{smallest color} \notin S$
  - $U \leftarrow U \setminus \{v\}$

**Bounds?**

# (Maximal) Independent Set

Independent Set: no two vertices share a common edge



# Parallel Graph Coloring

- Any independent set can be colored in parallel

- $U \leftarrow V$
- while $|U| > 0$ do in parallel
  - Choose an independent set $I$ from $U$
  - Color all vertices in $I$
  - $U \leftarrow U \setminus I$

- Optimal Coloring → color using smallest color
- Balanced Coloring → use all colors equally

# Maximal Independent Set (Luby)

- find largest MIS from graph
- Color all with the same color and remove from graph
- Recurse

$I \leftarrow \emptyset$

$V' \leftarrow V$

**while** $|V'| > 0$ **do**

choose and independent set $I'$ from $V'$

$I \leftarrow I + I'$

$X \leftarrow I' + N(I')$

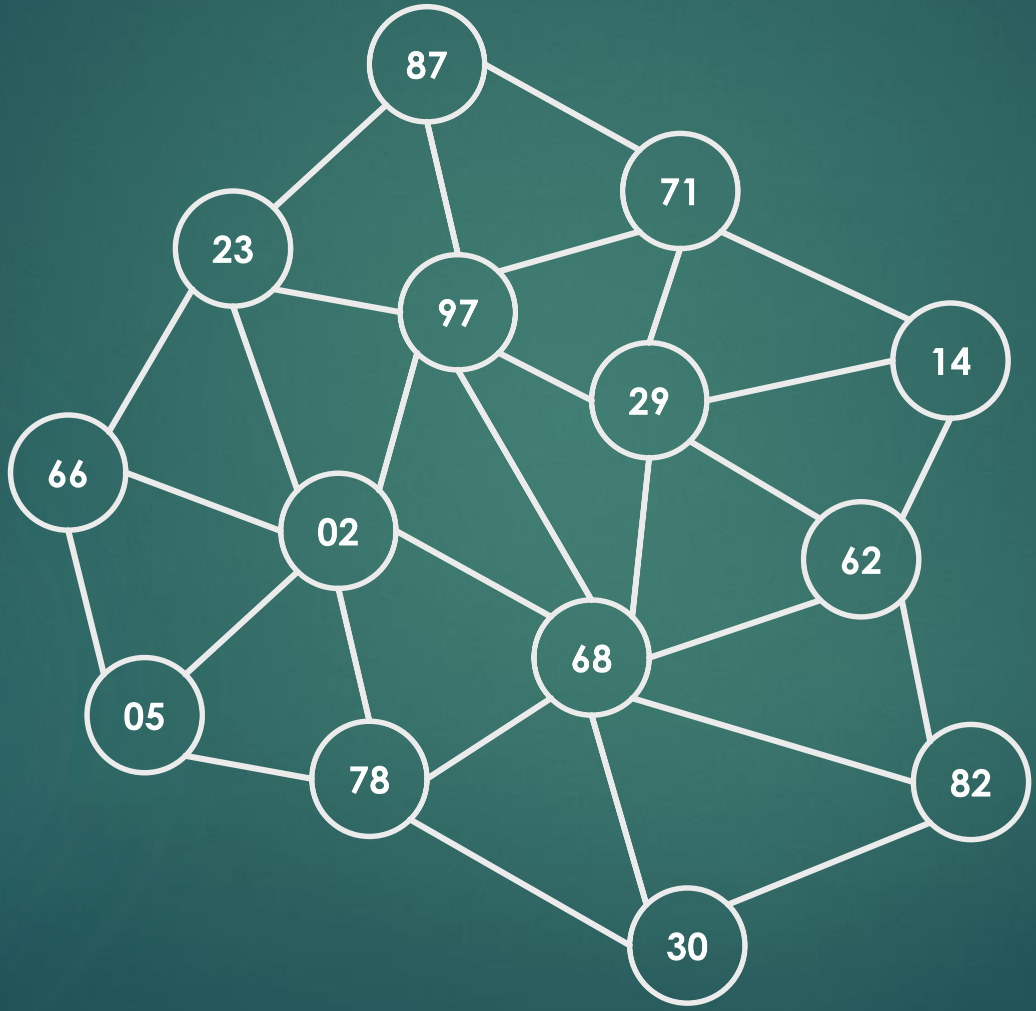
$V' \leftarrow V' - X$

# How to choose independent sets in parallel?

- Assign a random weight to each vertex
- Choose vertices that are a local maxima

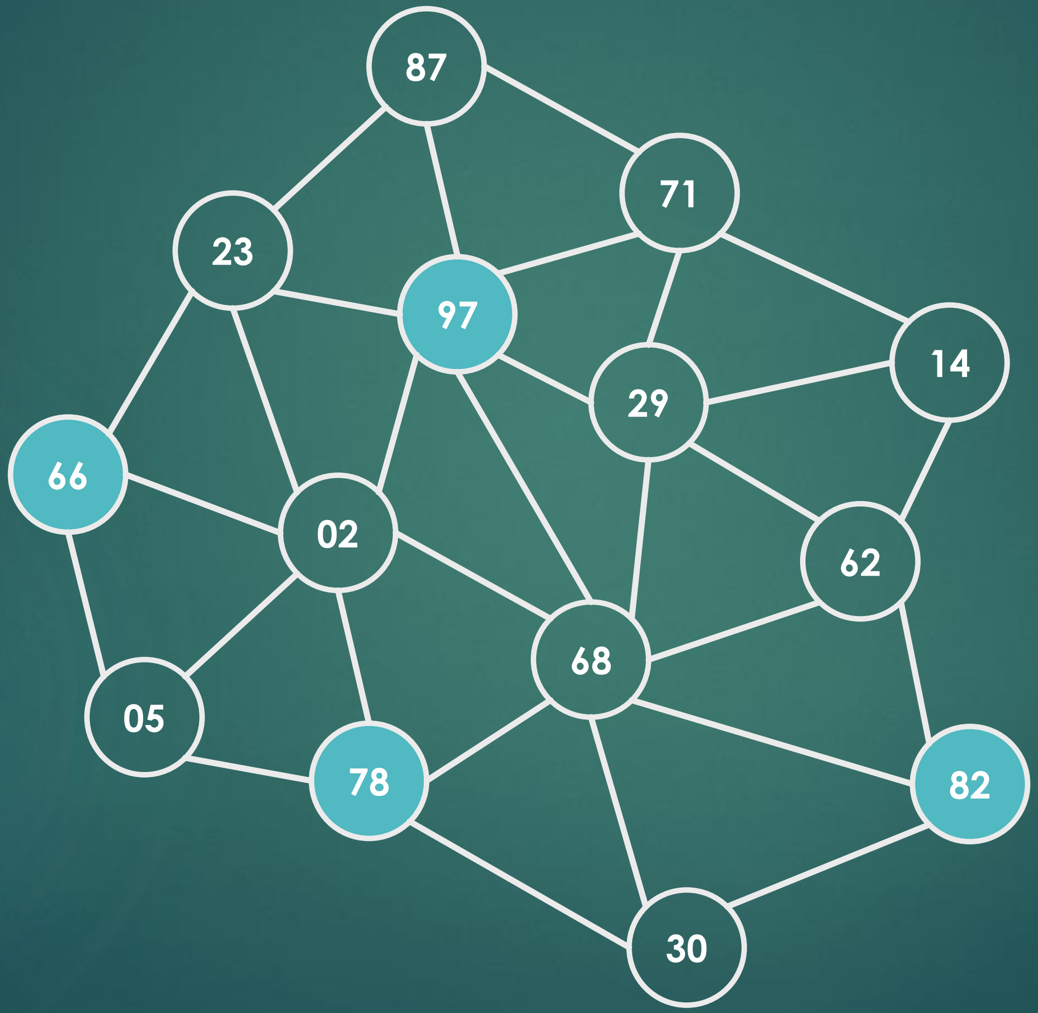
- $\mathcal{O}\big((c+1)\log|V|\big)$ algorithm
  - for sparse graphs

# Luby's Algorithm



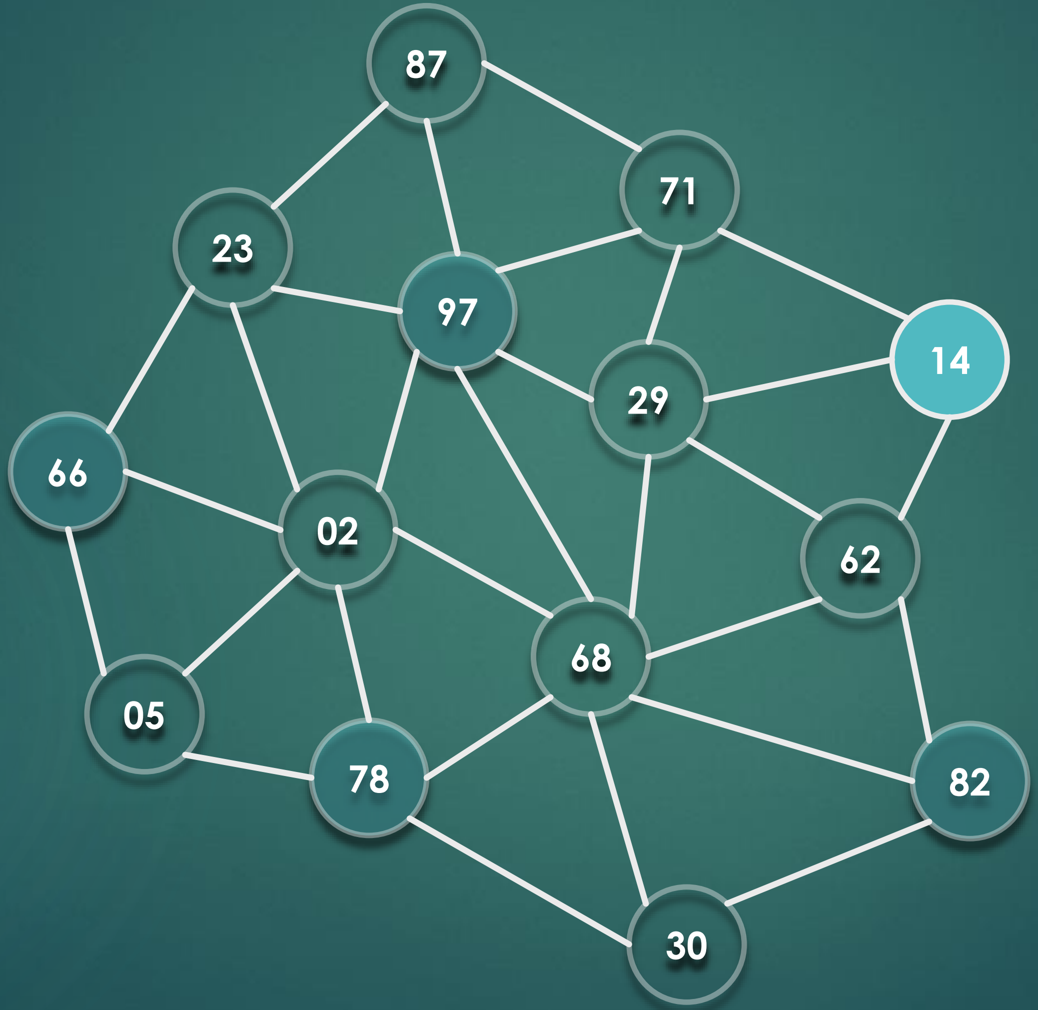

initially assigned random numbers

# Luby’s Algorithm



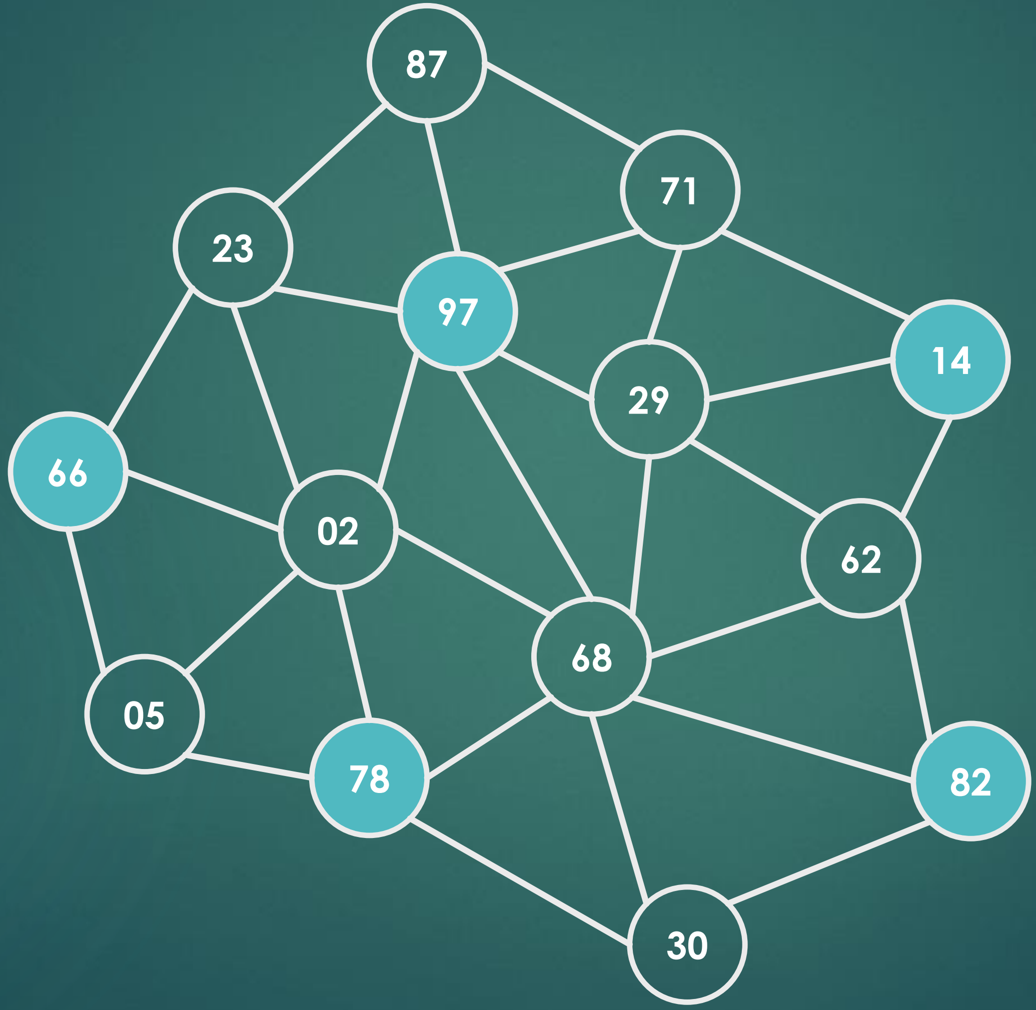

Find MIS

# Luby's Algorithm



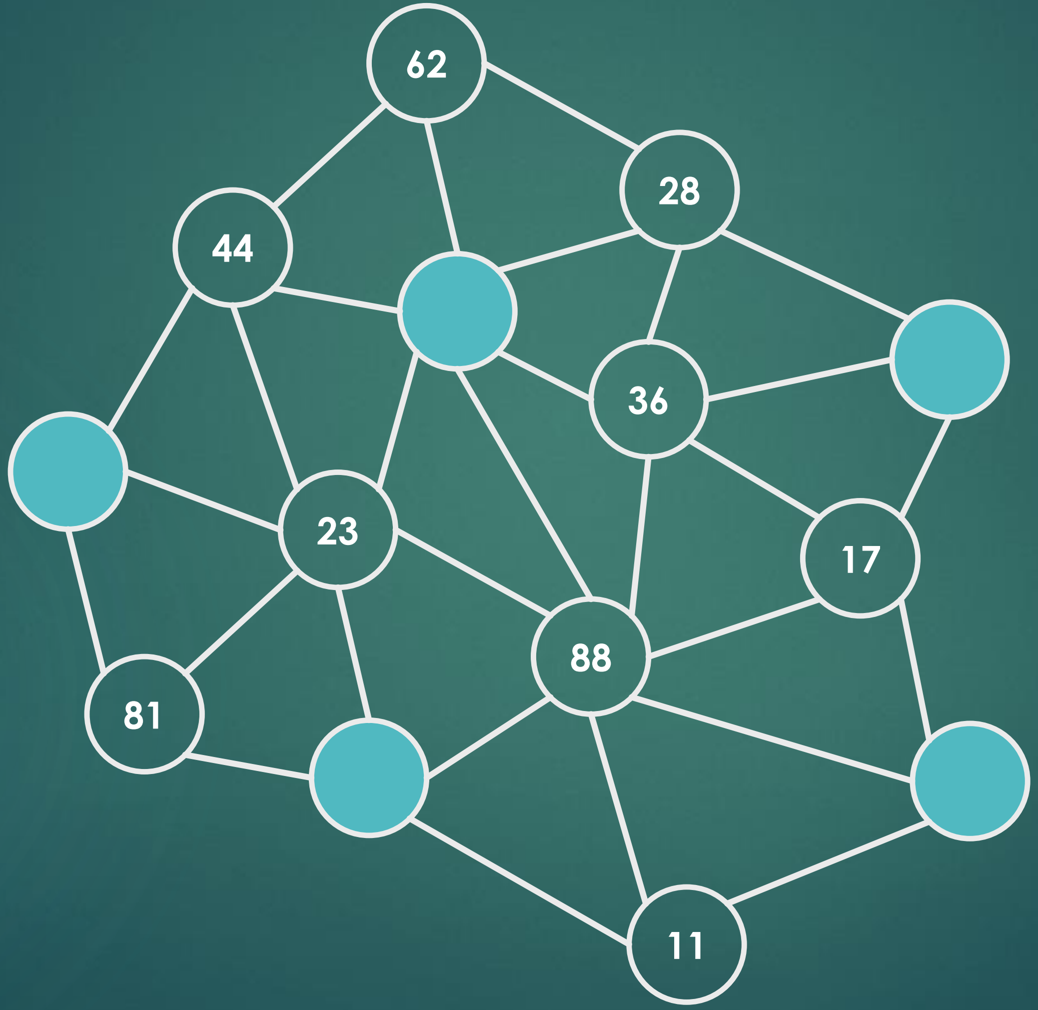

Find MIS

# Luby's Algorithm



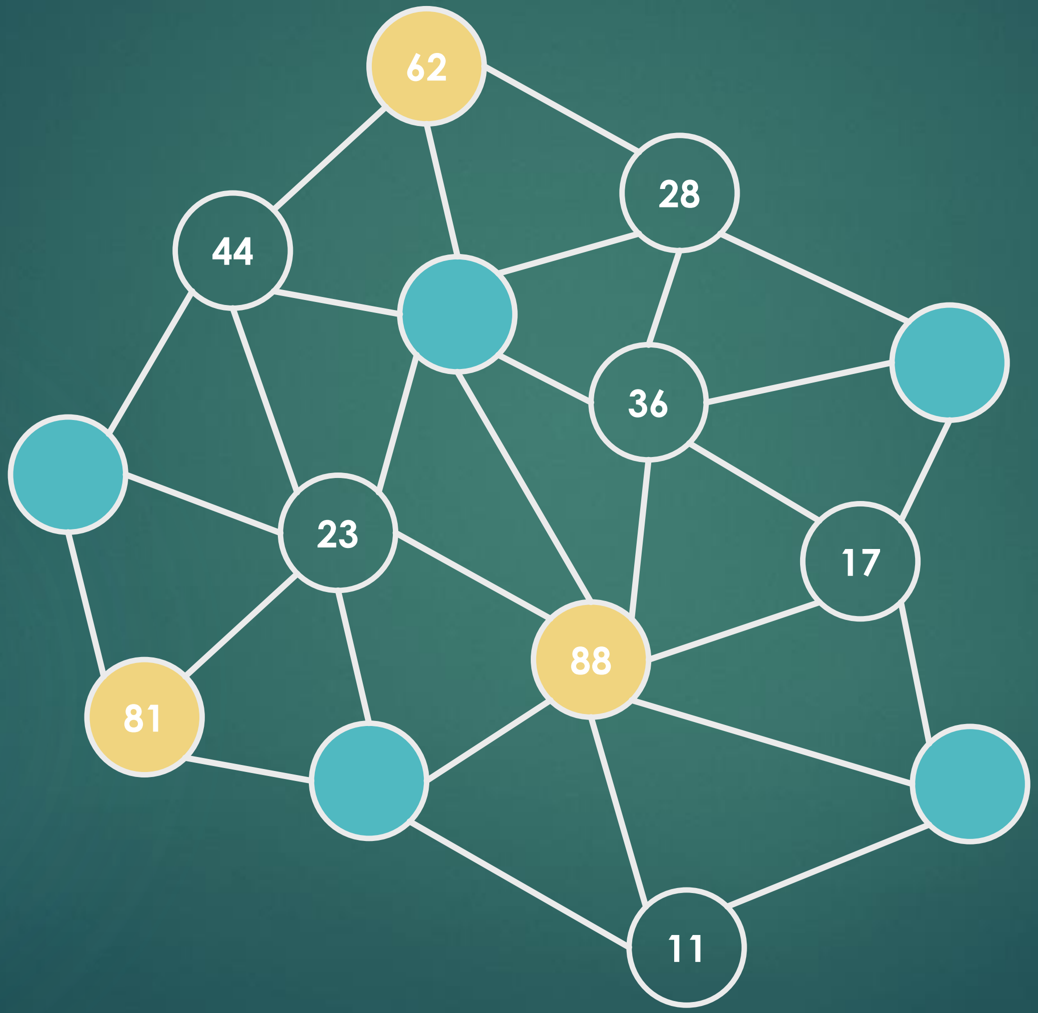

Find MIS, color all the same color

# Luby's Algorithm



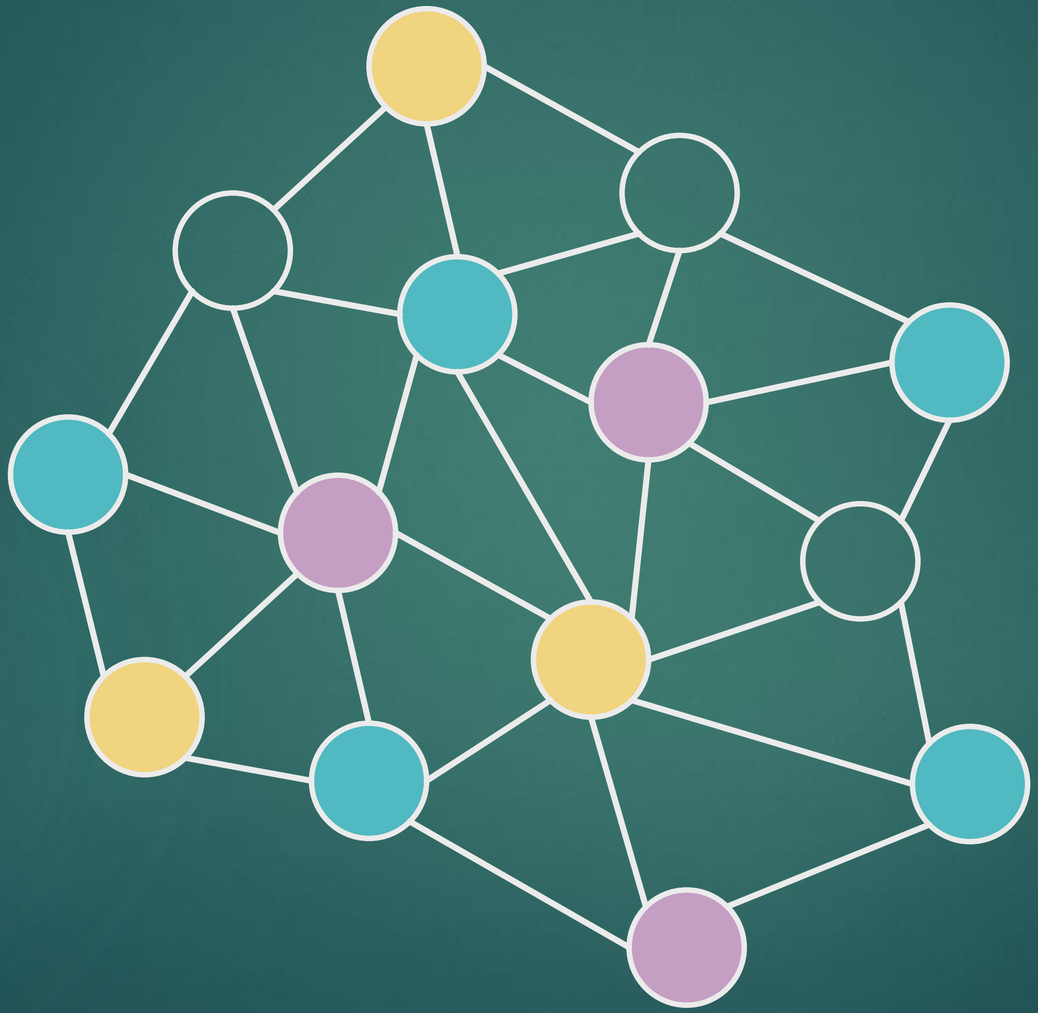

repeat

# Luby's Algorithm



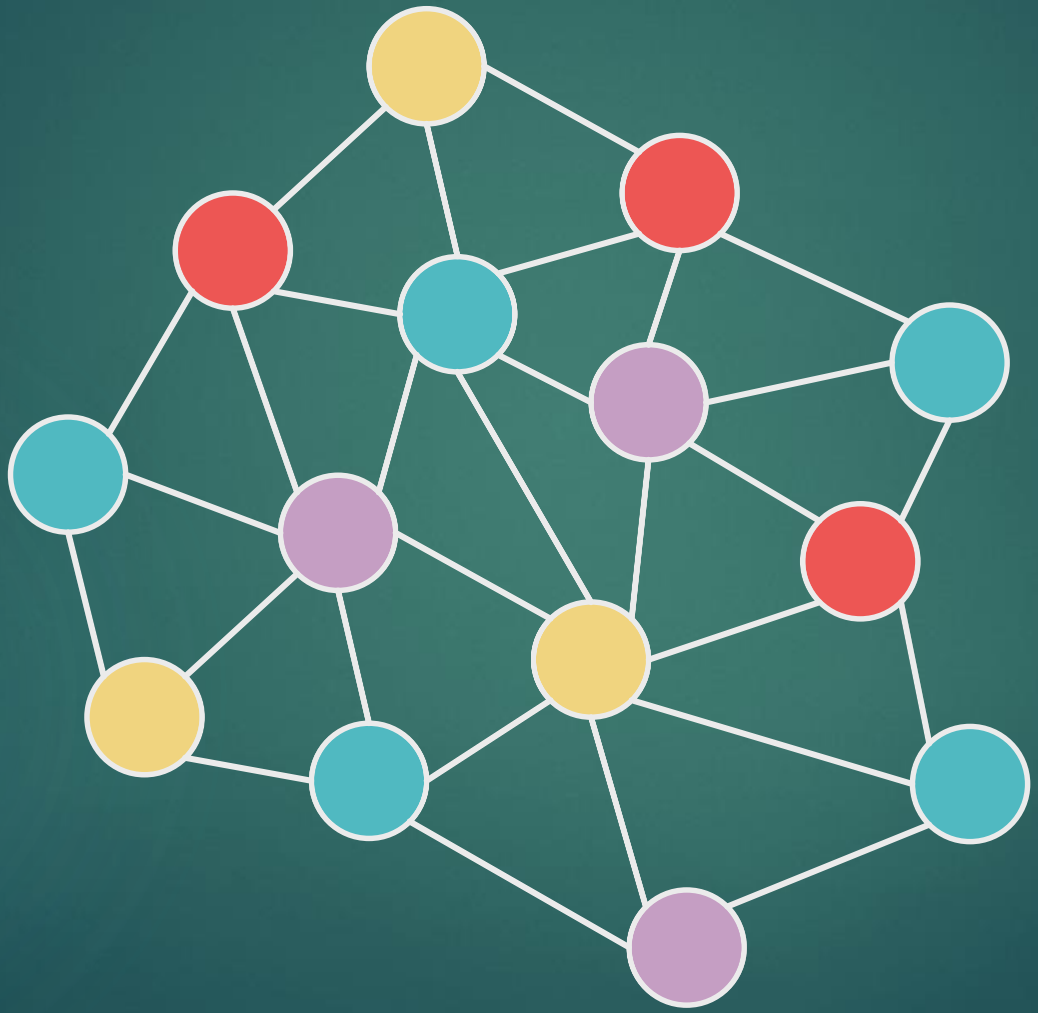

repeat

# Luby's Algorithm



repeat

# Luby's Algorithm



repeat

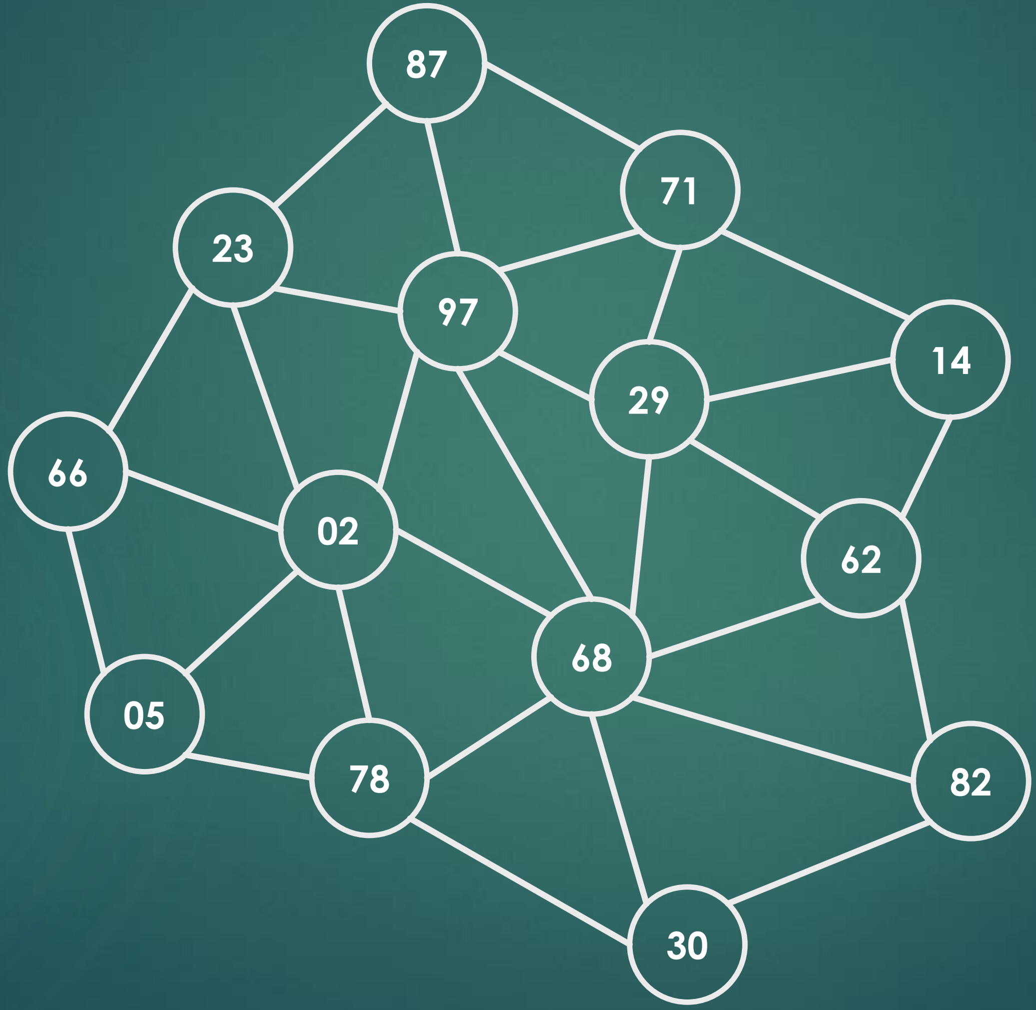
# Jones-Plassmann Coloring

- Not necessary to create a new random permutation of vertices every time
- Use vertex number to resolve conflicts
- Does not find a MIS at each step
- Instead,
  - Find independent set
  - Not assigned the same color
  - Color individually using smallest available color

# Jones-Plassmann Coloring

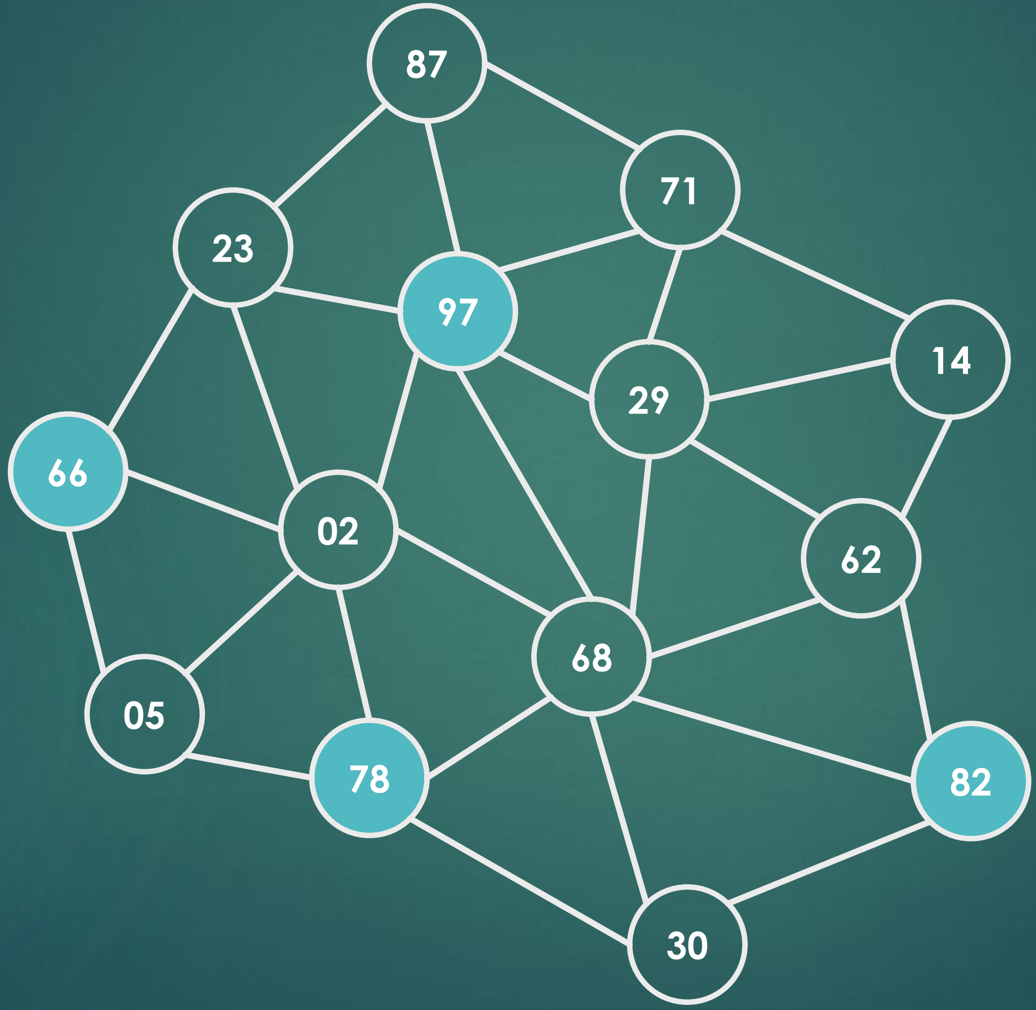
- $U \leftarrow V$
- **while** $|U| > 0$ **do**
  - **for all** vertices $v \in U$ **do in parallel**
    - $I \leftarrow \{v \mid w(v) > w(u)\ \forall\, u \in N(v)\}$
    - **for all** vertices $v' \in I$ **do in parallel**
      - $S \leftarrow \{\text{colors of } N(v')\}$
      - $c(v') \leftarrow \text{minimum color} \notin S$
  - $U \leftarrow U \setminus I$

# Jones-Plassmann Coloring



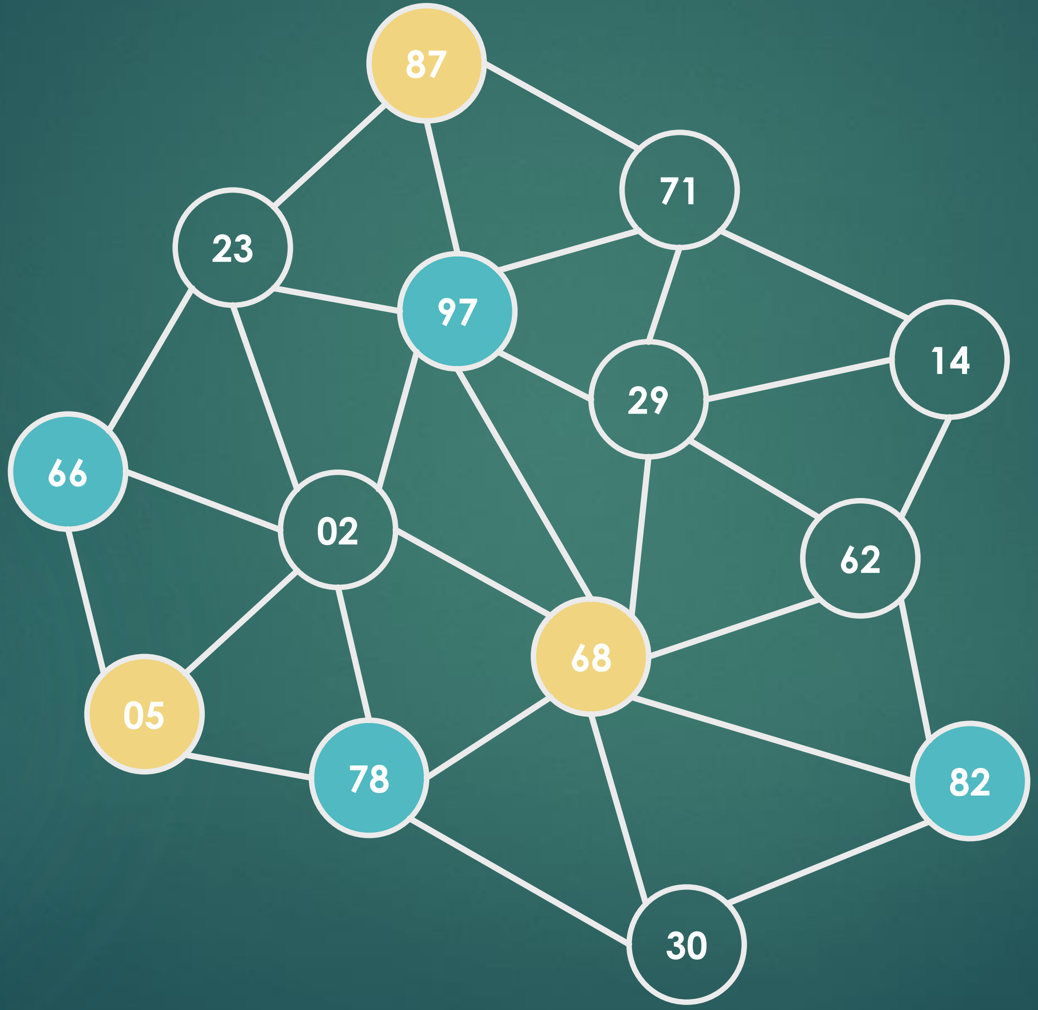

initially assigned random numbers

# Jones-Plassmann Coloring



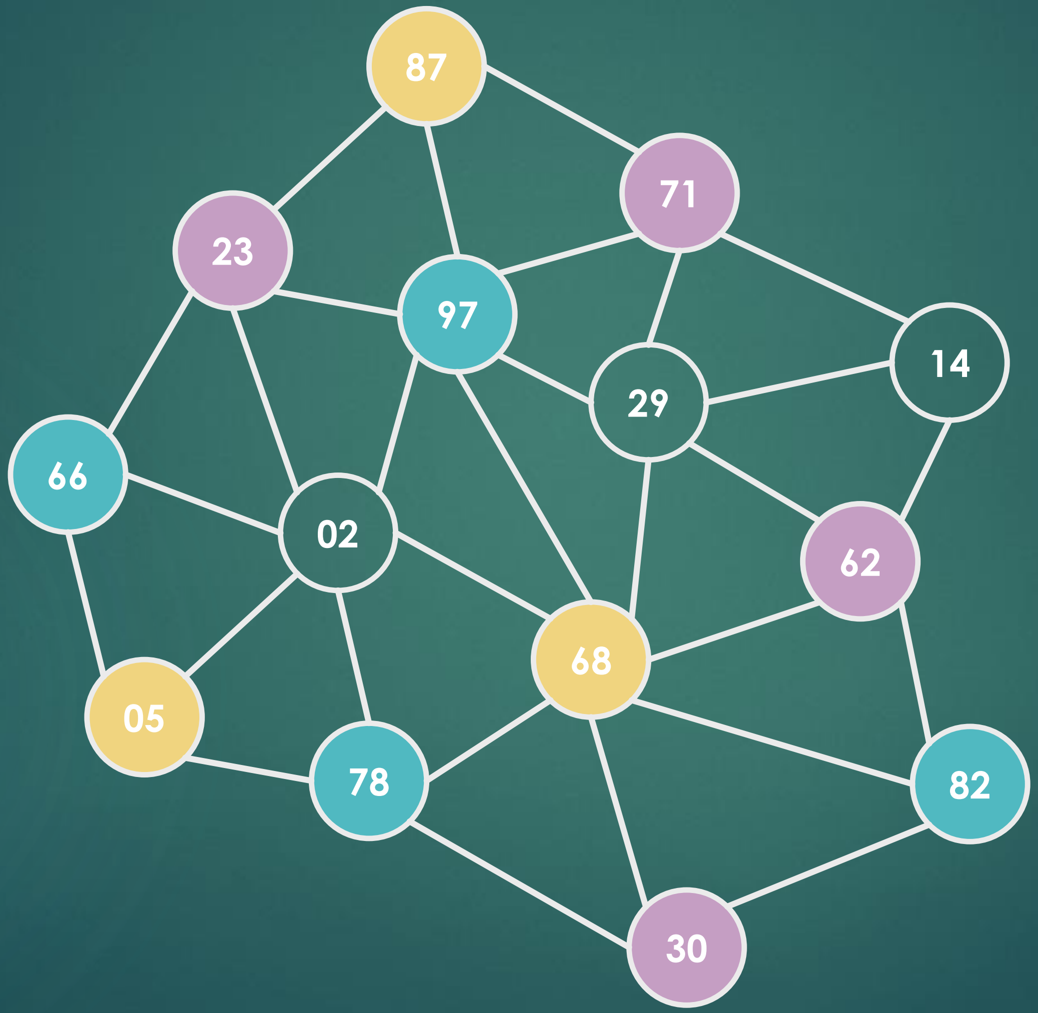

If local maxima, assign lowest color

# Jones-Plassmann Coloring



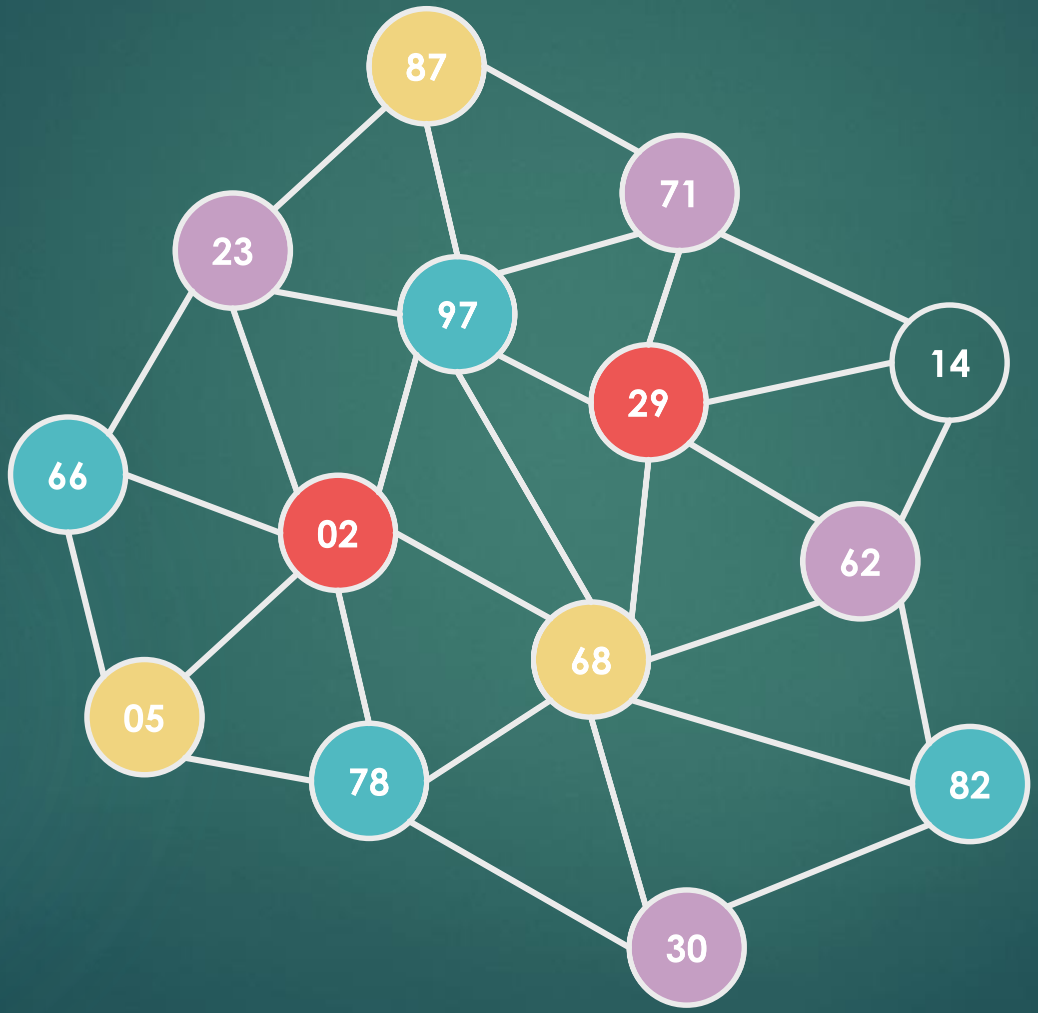

repeat, considering only uncolored vertices

# Jones-Plassmann Coloring



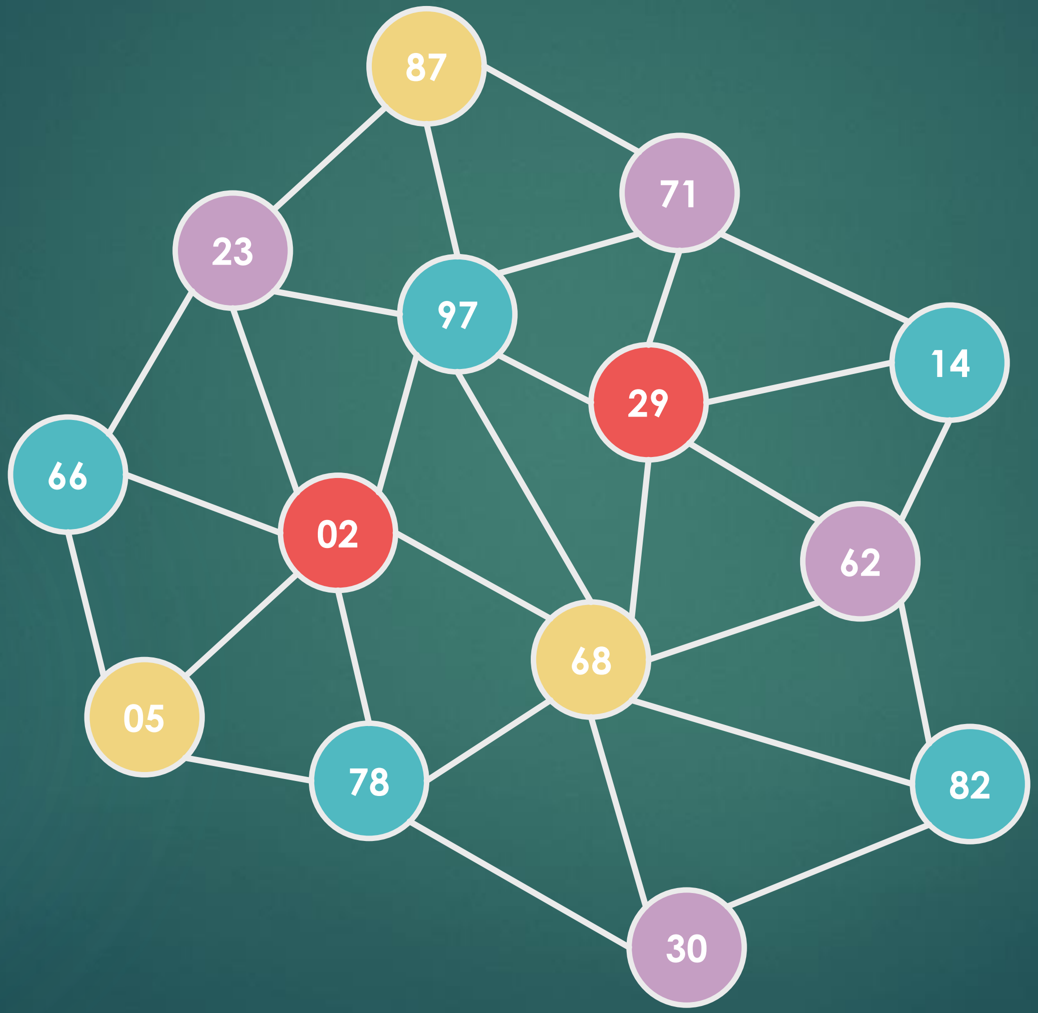

repeat, considering only uncolored vertices

# Jones-Plassmann Coloring




repeat, considering only uncolored vertices

# Jones-Plassmann Coloring




repeat, considering only uncolored vertices