Lecture 4: Graph Algorithms
Definitions

- **Undirected graph**: $G = (V, E)$
  - $V$ finite set of vertices, $E$ finite set of edges
  - any edge $e = (u, v)$ is an unordered pair
- **Directed graph**: edges are ordered pairs
- If $e = (u, v)$ is an edge in an undirected graph
  - $e$ is incident on vertices $u$ and $v$
- If directed graph
  - $e$ is incident from $u$ and incident into $v$
- If $e = (u, v)$ is an edge and $G$ undirected
  - $u$ and $v$ are adjacent to each other
- If $e = (u, v)$ is an edge and $G$ directed
  - $v$ is adjacent to $u$
Definitions

Undirected graph

Directed graph
Definitions

- **A path from u to v:** a sequence \( <v_0, v_1, v_2, \ldots, v_k> \) of vertices where \( v_0 = v \), \( v_k = u \) and \( (v_i, v_{i+1}) \) are edges.
- **Path length** = number of edges in the path
- **u reachable from v:** if there exists a path from \( v \) to \( u \)
- **Simple path:** if all of its vertices are distinct
- **Cycle:** if \( v_0 = v_k \)
- **Acyclic graph** = a graph without cycles
- **Simple cycle:** if all the intermediate vertices are distinct
- **Connected graph:** if every pair of vertices is connected by a path.
- \( G' = (V', E') \) is a **subgraph** of \( G = (V, E) \) if \( V' \) is a subset of \( V \) and \( E' \) is a subset of \( E \)
- **Complete graph:** if each pair of vertices is adjacent
- **Weighted graph:** each edge has an associated weight
Adjacency Matrix Representation

\[ A = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0
\end{bmatrix} \]

Lower bound for an algorithm (using adjacency matrix representation) needing to traverse all the edges = \( \Omega(|V|^2) \)

Space complexity = \( \Theta(|V|^2) \)
Adjacency List Representation

Adj[v] = list of vertices adjacent to v

Space complexity = \(\Theta(|E|)\)

Efficient when the graph is sparse (i.e. \(|E| \ll O(|V|^2)\))

Lower bound for an algorithm (using adjacency list representation) needing to traverse all the edges = \(\Omega(|V|+|E|) = \Omega(|E|)\)
Minimum Spanning Tree: Prim’s Algorithm

- *Spanning Tree of an undirected graph* $G$: a subgraph that is a tree containing all the vertices of $G$.
- *Minimum Spanning Tree (MST) for a weighted graph*: a spanning tree with minimum weight.
- If $G$ not connected it cannot have a spanning tree, it has a *spanning forest*.
- Determining the spanning forest $\Rightarrow$ apply MST algorithm for each connected component
- *Assumption*: we consider only connected graphs.
MST: Example

G

MST of G
Prim’s Algorithm

- **Greedy algorithm** i.e. it makes whatever is the best choice at the moment
- **Idea:** Selects an arbitrary starting vertex and then grows the tree by choosing a new vertex and edge that is guaranteed to be in the MST.
- **Notations:**
  - \( V_T \) = set that holds the vertices of the MST during its construction
  - \( d[v] \) = holds the weight of the edge with least weight from any vertex in \( V_T \) to vertex \( v \).
- **Complexity:** \( T_s = \Theta(n^2) \)
- Best implementation using priority queues \( O(E \log V) \)
Prim’s Algorithm

procedure PRIM_MST(V, E, w, r)
begin
  \( V_T := \{r\} \);
  \( d[r] := 0 \);
  for all \( v \in (V - V_T) \) do
    if edge \((r, v)\) exists set \( d[v] := w(r, v) \);
    else set \( d[v] := \infty \);
  while \( V_T \neq V \) do
  begin
    find a vertex \( u \) such that \( d[u] := \min\{d[v] | v \in (V - V_T)\} \);
    \( V_T := V_T \cup \{u\} \);
    for all \( v \in (V - V_T) \) do
      \( d[v] := \min\{d[v], w(u, v)\} \);
  endwhile
end PRIM_MST
Prim’s Algorithm: Example

(a) Original graph

(b) After the first edge has been selected

\[
\begin{array}{ccccccc}
\text{Adjacency Matrix} & a & b & c & d & e & f \\
\hline
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & \infty & 5 & 0 \\
\end{array}
\]
Prim’s Algorithm: Example

(c) After the second edge has been selected

(d) Final minimum spanning tree

\[
\begin{array}{ccccccc}
  & a & b & c & d & e & f \\
\hline
  a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & \infty & 5 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
  & a & b & c & d & e & f \\
\hline
  a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & \infty & 5 \\
\end{array}
\]
Prim’s Algorithm: Parallel Formulation

- It is hard to select more than one vertex to include in the MST
- The iterations of the while loop cannot be easily parallelized.
- **Parallelization:** partition $V$ into $p$ subsets using the 1-D block mapping.
- $V_i$ = the set of vertices assigned to $P_i$

**Steps:**
- $P_i$ computes
  $$d_i[u] = \min \{ d_i[v] \mid v \text{ belongs to } (V-V_T) \cap V_i \}$$
- All-to-one reduction to compute the minimum over all $d_i[u]$ at $P_0$
- $P_0$ inserts the new vertex $u$ into $V_T$ and broadcasts $u$ to all processes
- $P_i$ responsible for vertex $u$ marks it as belonging to $V_T$
- Each process updates the value of $d[v]$ for its local vertices
Prim’s Algorithm: Parallel Formulation

\[ d[1..n] \]

\[
\begin{array}{c|c|c|c}
\hline
\text{Processors} & 0 & 1 & i & p-1 \\
\hline
\end{array}
\]

\[ \frac{n}{p} \]
Parallel Prim’s Algorithm: Analysis

Steps: (executed n times)

- **$P_i$** computes
  \[ d_i[u] = \min \{d_i[v] \mid v \text{ belongs to } (V-V_T) \cap V_i \} \]
  => takes $\Theta(n/p)$

- All-to-one reduction to compute the minimum over all $d_i[u]$ at $P_0$
  => takes $\Theta(\log p)$

- $P_0$ inserts the new vertex $u$ into $V_T$ and broadcasts $u$ to all processes
  => takes $\Theta(\log p)$

- $P_i$ responsible for vertex $u$ marks it as belonging to $V_T$
  => takes $\Theta(1)$

- Each process updates the value of $d[v]$ for its local vertices
  => takes $\Theta(n/p)$

- $T_p = \Theta(n^2/p) + \Theta(n \log p)$

- Cost optimal if $(p \log p)/n = O(1) \Rightarrow p = O(n/\log n)$

- Isoefficiency function $\Theta(p^2 \log^2 p)$
Single-Source Shortest Paths: Dijkstra’s Algorithm

- Edsger W. Dijkstra (1930-2002)
  http://www.cs.utexas.edu/users/EWD/
- *Single-source shortest paths problem:* find the shortest paths from a vertex $v$ to all other vertices in $V$
- *Shortest path* = minimum weight path
- *Dijkstra’s Algorithm* (1959) solves the problem on both directed and undirected graphs with non-negative weights.
- Similar to Prim’s algorithm
- *Greedy algorithm:* it always chooses an edge to a vertex that appears closest.
- $l[u] =$ minimum cost to reach vertex $u$ from vertex $s$ by means of vertices in $V_T$. 
Single-Source Shortest Paths:
Dijkstra’s Algorithm

procedure DIJKSTRA_SINGE_SOURCE_SP(V, E, w, s)
begin
    \( V_T := \{s\}; \)
    for all \( v \in (V - V_T) \) do
        if \( (s, v) \) exists set \( l[v] := w(s, v); \)
        else set \( l[v] := \infty; \)
    while \( V_T \neq V \) do
        begin
            find a vertex \( u \) such that \( l[u] := \min\{l[v] \mid v \in (V - V_T)\}; \)
            \( V_T := V_T \cup \{u\}; \)
            for all \( v \in (V - V_T) \) do
                \( l[v] := \min\{l[v], l[u] + w(u, v)\}; \)
        endwhile
    end DIJKSTRA_SINGE_SOURCE_SP

\( T_s = \Theta(n^2) \)
Example: Dijkstra’s Single-Source Shortest-Paths Algorithm
Parallel Dijkstra’s Algorithm: Analysis

- Same as Prim’s algorithm
- $T_p = \Theta(n^2/p) + \Theta(n \log p)$
- Cost optimal if $(p \log p)/n = O(1) \Rightarrow p = O(n/\log n)$
- Isoefficiency function $\Theta(p^2 \log^2 p)$
All-Pairs Shortest Paths

- **All-pairs shortest paths problem**: find the shortest paths between all pairs of vertices $v_i$, $v_j$ such that $i \neq j$.
- **Output**: $nxn$ matrix $D = (d_{i,j})$ such that $d_{i,j}$ is the cost of the shortest path from $v_i$ to $v_j$.
- **Dijkstra’s Algorithm**: for graphs with non-negative weights
  
  **Idea**: Apply Dijkstra’s Algorithm for each vertex $\Rightarrow T_s = \Theta(n \times n^2)$

- **Floyd’s Algorithm**: for graphs having negative weights but no negative-weight cycles.
Dijkstra’s Algorithm: Parallel Formulation

- Two parallel formulations:
  - **Source-partitioned formulation**: partition the vertices among different processes and have each process compute the single-source shortest path for all vertices assigned to it.
  - **Source-parallel formulation**: assign each vertex to a set of processes and use the parallel formulation of the single-source shortest path algorithm.
Source-Partitioned Formulation

- Uses $n$ processes
- Each process $P_i$ finds the shortest paths from vertex $v_i$ to all other vertices by executing the sequential Dijkstra’s algorithm locally.
- If adjacency matrix is replicated at each process $\Rightarrow$ no communication
- $T_p = \Theta(n^2)$
- $S = \Theta(n^3)/\Theta(n^2) = \Theta(n)$
- $E = \Theta(1) \Rightarrow$ very good algorithm!
- Isoefficiency: at most $n$ processes can be used $\Rightarrow p = n \Rightarrow W = \Theta(n^3) = \Theta(p^3)$ not very scalable!
Source-Parallel Formulation

- Uses parallel Dijkstra for each vertex => $p > n$
- $p$ processes are divided into $n$ partitions each with $p/n$ processes.
- Each partition solves one single-source shortest path problem.
- *Total number of processes* that can be used efficiently => $O(n^2)$
- $T_p = \Theta(n^2/(p/n)) + \Theta(n \log(p/n)) = \Theta(n^3/p)+\Theta(n \log p)$
- *Cost optimal* if $p \log p = O(n^2) => p \log n = O(n^2)$
  => $p = O(n^2/\log n)$
- *Isoefficiency due to communication*: $\Theta((p \log p)^{1.5})$
- *Isoefficiency due to concurrency*: $\Theta(p^{1.5})$
  (because $p = O(n^2)$)
- *Overall isoefficiency*: $\Theta((p \log p)^{1.5})$
- More scalable than source-partitioned formulation
- It exploits more parallelism!
Floyd’s Algorithm

• Robert W Floyd (1936-2001)
• “Bob used to say that he was planning to get a Ph.D. by the “green stamp method,” namely by saving envelopes addressed to him as ‘Dr. Floyd’. After collecting 500 such letters, he mused, a university somewhere in Arizona would probably grant him a degree.” – D. Knuth (sigact.acm.org/floyd/)

• Observations:
  – \( A_k = \{v_1, v_2, \ldots, v_k\} \) a subset of vertices for \( k \leq n \)
  – \( p_{i,j}(k) \) the minimum weight path from \( v_i \) to \( v_j \) whose intermediate vertices belong to \( A_k \)
  – \( d_{i,j}(k) = \text{weight of } P_{i,j}(k) \)
  – If \( v_k \) is not on the shortest path from \( v_i \) to \( v_j \) then \( p_{i,j}(k)=p_{i,j}(k-1) \)
  – If \( v_k \) is in \( p_{i,j}(k) \) then break \( p_{i,j}(k) \) into two paths \( v_i \rightarrow v_k \) and \( v_k \rightarrow v_j \). These paths use vertices from \( \{v_1, v_2, \ldots, v_{k-1}\} \Rightarrow \)
    \[ d_{i,j}(k) = d_{i,k}(k-1) + d_{k,j}(k-1) \]

• Recurrence relation:

\[
d_{i,j}(k) = \begin{cases} 
    w(v_i, v_j) & \quad k = 0 \\
    \min\{d_{i,j}(k-1), d_{i,k}(k-1) + d_{k,j}(k-1)\} & \quad k \geq 1
\end{cases}
\]

• The solution is given by \( D(n) \)
Floyd’s Algorithm

procedure FLOYD_ALL_PAIRS_SP(A)
begin
    \( D^{(0)} = A \);
    for \( k := 1 \) to \( n \) do
        for \( i := 1 \) to \( n \) do
            for \( j := 1 \) to \( n \) do
                \( d_{i,j}^{(k)} := \min \left( d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right) \);
    \end FLOYD_ALL_PAIRS_SP

\( T_s = \Theta(n^3) \)
Space complexity = \( \Theta(n^2) \)
Floyd’s Algorithm: Example

\[
\begin{align*}
D(0) & = \begin{bmatrix}
0 & 8 & 5 \\
3 & 0 & \infty \\
\infty & 2 & 0
\end{bmatrix} &
D(1) & = \begin{bmatrix}
0 & 8 & 5 \\
3 & 0 & 8 \\
\infty & 2 & 0
\end{bmatrix} \\
D(2) & = \begin{bmatrix}
0 & 8 & 5 \\
3 & 0 & 8 \\
5 & 2 & 0
\end{bmatrix} &
D(3) & = \begin{bmatrix}
0 & 8 & 5 \\
3 & 0 & 8 \\
5 & 2 & 0
\end{bmatrix}
\end{align*}
\]
Floyd’s Algorithm: Parallel Formulation

- Partition matrix $D(k)$ using 2-D block partitioning
- Each $(n/p^{1/2} \times n/p^{1/2})$ block is assigned to one process

\[
\begin{array}{c|c|c}
\frac{n}{\sqrt{p}} & \frac{n}{\sqrt{p}} & \frac{n}{\sqrt{p}} \\
(1,1) & (1,2) & \\
(2,1) & & \\
\end{array}
\]

\[(i - 1) \cdot \frac{n}{\sqrt{p}} + 1, (j - 1) \cdot \frac{n}{\sqrt{p}} + 1\]
Floyd’s Algorithm: Parallel Formulation

- Each process updates its part of the matrix during each iteration
- During the $k$-th iteration each process needs certain segments of the $k$-th row and $k$-th column
- Idea: broadcast row-wise and column-wise the corresponding segments
Floyd’s Algorithm:
Parallel Formulation

procedure FLOYD_2DBLOCK(D(0))
begin
   for k := 1 to n do
      begin
         each process Pᵢ,j that has a segment of the kᵗʰ row of D(ᵏ⁻¹);
         broadcasts it to the Pᵢ,∗ processes;
         each process Pᵢ,j that has a segment of the kᵗʰ column of D(ᵏ⁻¹);
         broadcasts it to the Pᵢ,∗ processes;
         each process waits to receive the needed segments;
         each process Pᵢ,j computes its part of the D(ᵏ) matrix;
      end
end FLOYD_2DBLOCK
Parallel Floyd’s Algorithm: Analysis

- **Assumption:** bisection bandwidth = $\Theta(p)$
- **Broadcast** of $n/p^{1/2}$ elements to $p^{1/2}-1$ processes
  $\Rightarrow \Theta(n/p^{1/2} \log p)$
- **Synchronization step** $\Rightarrow \Theta(\log p)$
- **Local computation** of $D(k)$ $\Rightarrow \Theta(n^2/p)$
- $T_p = \Theta(n^3/p) + \Theta(n^2/p^{1/2} \log p)$
- **Cost optimal** if $(p^{1/2} \log p)/n = O(1)$ $\Rightarrow p = O(n^2/\log^2 n)$
- **Isoefficiency due to communication:** $\Theta(p^{1.5} \log^3 p)$
- **Isoefficiency due to concurrency:** $\Theta(p^{1.5})$
- **Overall isoefficiency:** $\Theta(p^{1.5} \log^3 p)$
- The overhead due to communication can be improved if the execution of the algorithm is pipelined $\Rightarrow$ the most scalable algorithm for all-pairs shortest paths.
Transitive Closure

- **Problem:** determine if two vertices in a graph are connected
- **Transitive closure of G:**
  the graph $G^* = (V, E^*)$ where
  $E^* = \{(v_i, v_j) \mid \text{there is a path from } v_i \text{ to } v_j \text{ in } G\}$
- **Connectivity matrix $A^* = (a^*_{i,j})$$**
  
  \[
  a^*_{i,j} = \begin{cases} 
  1 & \text{if there is a path from } v_i \text{ to } v_j \text{ or } i = j \\
  \infty & \text{otherwise}
  \end{cases}
  \]
- **Method 1:** assign weights of 1 to each edge and use any of the all-pairs shortest paths algorithms and obtain matrix $A$ from $D$
  
  \[
  a^*_{i,j} = \begin{cases} 
  1 & \text{if } d_{i,j} > 0 \text{ or } i = j \\
  \infty & \text{if } d_{i,j} = \infty
  \end{cases}
  \]
- **Method 2:** use Floyd’s algorithm on the adjacency matrix replacing the “min” and “+” by “or” and “and”

  Initially: $a_{i,j} = 1$ if $i = j$ or $(v_i, v_j)$ is an edge of $G$ and $a_{i,j} = 0$ otherwise
  
  Matrix $A^*$: $a^*_{i,j} = \infty$ if $d_{i,j} = 0$ and $a^*_{i,j} = 1$ otherwise
Transitive Closure: Example

$G$  

Transitive closure $G^*$
Connected Components

- **Connected components of undirected graph** $G$: the maximal disjoint sets $C_1, C_2, \ldots, C_k$ such that $V = C_1 \cup C_2 \cup \ldots \cup C_k$ and $u, v$ in $C_i \Leftrightarrow u$ is reachable from $v$ and $v$ is reachable from $u$
- $C_i$ are equivalence classes of vertices under “is reachable from” relation

Graph with 3 connected components
Depth-First Search (DFS) Based Algorithm

• **Idea:** Search “deeper’ in the graph

• **DFS Algorithm:**
  – Edges are explored out of the most recently discovered vertex \( v \) that still has unexplored edges leaving \( v \).
  – When all of \( v \)’s edges have been explored, the search backtracks to explore edges leaving the vertex from which \( v \) was discovered.
  – The process continues until we have discovered all vertices reachable from the original source.
  – If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated.
  – The process is repeated until all vertices are discovered.

• \( T_s = \Theta(|E|) \)
Depth-First Search (DFS) Based Algorithm

- **A**: unexplored vertex
- **A**: visited vertex
- **unexplored edge**
- **discovery edge**
- **back edge**
Depth-First Search (DFS) Based Algorithm
Using DFS to Find Connected Components

(a) $G$

(b) Connected components of $G$
Connected Components: Parallel Formulation

- Partition the adjacency matrix of $G$ into $p$ parts and assign each part to one of $p$ processes
- $P_i$ has subgraph $G_i$

**Algorithm:**
- **First step:** Each $P_i$ computes the dept-first forest of $G_i$ using DFS algorithm $\Rightarrow$ $p$ spanning forests
- **Second step:** Pairwise merging of the $p$ spanning forests into one spanning forest

- How to merge the forests?
- **Solution:** use disjoint sets operations
Connected Components: Parallel Formulation

- **Disjoint sets operations:**
  - \( \text{Find}(x) \) returns a pointer to the representative element of the set containing vertex \( x \) (unique to each set)
  - \( \text{Union}(x,y) \) unites the sets containing elements \( x \) and \( y \)

- **Algorithm:**
  - Initially each edge is a set
  - \textbf{for} each edge \((u,v)\) in \( E \)
    - \textbf{if} \( \text{Find}(u) \neq \text{Find}(v) \) \textbf{then} \( \text{Union}(u,v) \)

- **Complexity:** \( O(n) \)

  For two forests => at most \( n-1 \) edges of one forest are merged to the edges of the other
Parallel Connected Components: Example
Parallel CC using 1-D Block Mapping

- The adjacency matrix is partitioned into $p$ stripes
- Assume a $p$ process message passing system
- **Step 1**: computing the spanning forest for graphs with $(n/p) \times n$ adjacency matrix
  \[ \Theta(n^2/p) \]
- **Step 2**: pairwise merging
  Performed by embedding a virtual tree on the processes
  \[ \log p \text{ merging stages each taking } \Theta(n) = \Theta(n \log p) \]
  Communication time => Spanning forests (\(\Theta(n)\) edges) are sent between nearest neighbors
  \[ \Theta(n \log p) \]

- \[ T_p = \Theta(n^2/p) + \Theta(n \log p) \]
- Cost optimal if $p = O(n/\log n)$
- Isoefficiency due to communication and extra computation: \[ \Theta(p^2 \log^2 p) \]
- Isoefficiency due to concurrency: \( \Theta(p) \)
- Overall isoefficiency: \( \Theta(p^2 \log^2 p) \)
  \[ \text{=> same as Prim’s algorithm and Dijkstra’s algorithm} \]
Algorithms for Sparse Graphs

- **Sparse graphs**: \(|E| \ll |V|^2\)
- Algorithms based on the *adjacency list representation*
- *Lower bound* => \(\Omega(|V| + |E|)\)
- The lower bound depend on the sparseness of the graph.
- Difficult to achieve even work distribution and low communication overhead for sparse graphs
- Possible partitioning methods:
  - Assign an *equal number of vertices* and their adjacency list to each process => may lead to significant load imbalance
  - Assign *equal number of edges* to each process => may require splitting the adjacency list of a vertex => communication overhead increases
- Hard to derive efficient parallel formulations for general sparse graphs
- Efficient parallel formulations for some structured sparse graphs
Sparse Graphs

Linear graph

Grid graph

Random sparse graph
Maximal Independent Set (MIS)

- A set of vertices $I$ is called independent if no pair of vertices in $I$ is connected via an edge in $G$.
- $I$ is a maximal independent set if by including any other vertex not in $I$, the independence property is violated.

\{a, d, i\} is an independent set
\{a, c, j, f, g\} is a maximal independent set
\{a, d, h, f\} is a maximal independent set
MIS: Simple Algorithm

- **Initially** \( I \) is empty and the set of candidates \( C = V \)
- **Algorithm:**
  
  while (\( C \) not empty)
  
  {
    move a vertex \( v \) from \( C \) into \( I \);
    remove all vertices adjacent to \( v \) from \( C \);
  }

- **Correctness:**
  
  - \( I \) is an *independent* set because all the vertices whose subsequent inclusion will violate the condition are removed from \( C \)
  
  - \( I \) is *maximal* because any other vertex that is not in \( I \) is adjacent to at least one of the vertices in \( I \).

- *Inherently serial!*
MIS: Luby’s Algorithm

- *Initially* $I$ is empty and the set of candidates $C = V$
- *Algorithm:*
  while (C not empty)
  {
    - Assign a unique random number to each vertex in $C$
    - If a vertex has a random number smaller than all of
      the random numbers of the adjacent vertices, it is
      included in $I$.
    - Update $C$ by removing the vertices included in $I$ and
      their adjacent vertices.
  }
- *Suitable for parallelization!"
MIS: Luby's Algorithm

(a) After the 1st random number assignment

(b) After the 2nd random number assignment

(c) Final maximal independent set

- **Vertex in the independent set**
- **Vertex adjacent to a vertex in the independent set**
Luby’s Algorithm: Shared Address Space Formulation

- \( I[i] = 1 \) if \( v_i \) is part of the MIS, or 0 otherwise
- \( C[i] = 1 \) if vertex \( i \) is part of the candidate set, or 0 otherwise
- \( R[i] \) stores the random number assigned to vertex \( v_i \)
- During each iteration the vector \( C \) is partitioned among the \( p \) processes
- Each process:
  - Generates a random number for its assigned vertices from \( C \)
  - Wait for all processes to generate the random numbers
  - For each vertex assigned to it checks to see if it can be included in \( I \). If yes, set the corresponding entry of \( I \) to 1.
  - Updates \( C \).
- Concurrent writes of zero into \( C \) do not affect the correctness (same value is written).
Single-Source Shortest Paths: Johnson’s Algorithm

- A variant of Dijkstra’s algorithm
- Uses a *priority queue* $Q$ to store the value $l[v]$ for each vertex in $V \setminus V_T$
- In the priority queue the element with the smallest value in $l$ is always at the front
  => implemented as a *min-heap*
  => $O(\log n)$ time for update.
- Overall complexity => $O(|E| \log n)$
Single-Source Shortest Paths: Johnson’s Algorithm

procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s) begin
    Q := V;
    for all v ∈ Q do
        l[v] := ∞;
        l[s] := 0;
    while Q ≠ ∅ do
        begin
            u := extract_min(Q);
            for each v ∈ Adj[u] do
                if v ∈ Q and l[u] + w(u, v) < l[v] then
                    l[v] := l[u] + w(u, v);
        endwhile
    end JOHNSON_SINGLE_SOURCE_SP
Johnson’s Algorithm: Parallelization Strategy

• **Problem:** How to maintain the priority queue efficiently in a parallel implementation?

• **Simple strategy:**
  – $P_0$ maintains the queue
  – The other processes compute new values for $l[v]$ and send them to $P_0$ to update $Q$

• **Limitations:**
  – A single process updates $Q$ => $O(|E| \log n)$ time
    => *No asymptotic speedup!*
  – During each iteration it updates roughly $|E|/|V|$ vertices
    => *No more than $|E|/|V|$ processes can be kept busy*

How to alleviate these limitations?
Johnson’s Algorithm: Parallelization Strategy

- **First limitation:**
  - Distribute the maintenance of Q to multiple processes => non-trivial
  - Achieves a small speedup of $O(\log n)$ (assuming one update takes $O(1)$)

- **Second limitation:**
  - Extract more than one vertex from Q at the same time.
  - If $v$ at the top of Q, extract all vertices $u$ that have $l[u] = l[v]$.
  - If we know that the minimum weight over all the edges is $m => extract all vertices such that $l[u] <= l[v] + m$ (these are called **safe vertices**, i.e. the update will not modify the next min in Q).

- **Complicated algorithm with limited concurrency.**
Johnson’s Algorithm: Parallelization Strategy

• **Alternate approach:**
  – Design a parallel algorithm that *processes both safe and unsafe vertices concurrently* as long as these safe edges can be reached from the source via a path involving vertices whose shortest paths have already been computed.
  – Each of the *p* processes extracts one of the top *p* vertices from *Q* and updates the value of adjacent vertices.

• Does not ensure that the *l* value of the vertices extracted from the priority queue corresponds to the cost of the shortest path.

• **Solution:** detect when we have incorrectly computed the shortest path to a vertex and insert it back into *Q* with the updated *l* value.

• *u* just extracted, *v* has already been extracted, *u* adjacent to *v* if
  \[ l[v] + w(v,u) \leq l[u] \]
  then the shortest path to *u* incorrectly computed => insert back *u* with
  \[ l[u] = l[v] + w(u,v) \]
Johnson’s Algorithm: Parallelization Strategy

Example: processing unsafe vertices concurrently

P₀: \( l[h] + w(h, g) = 5 < l[g] = 10 \) (from previous iteration)

=> insert \( g \) back into \( Q \) with \( l[g] = 5 \)
Johnson’s Algorithm: Distributed Memory Formulation

- **Idea**: remove the bottleneck of working with a single priority queue.
- \( V \) is partitioned among \( p \) processes
- \( P_i \) has:
  - a local priority queue \( Q_i \)
  - an array \( sp[] \) that store the shortest path from source to the vertices assigned to \( P_i \)
- \( sp[v] \) is updated from \( l[v] \) each time \( v \) is extracted from the queue.
- Initially, \( sp[v] = \infty \), \( sp[s] = 0 \)
- *Each process executes Johnson’s algorithm locally.*
- At the end \( sp[v] \) stores the shortest path from source to vertex \( v \).
Johnson’s Algorithm: Distributed Memory Formulation

- How to maintain the queues?
- Assume \( (u,v) \) is an edge, \( P_i \) has just extracted \( u \) from \( Q_i \)
- \( P_i \) sends \( l[u] + w(u,v) \) to \( P_j \)
- \( P_j \) receives the message and sets the values of \( l[v] \) in \( Q_j \) to \( \min\{l[v], l[v] + w(u,v)\} \)
- \( P_j \) might have already computed \( sp[v] \) => two cases:
  - If \( sp[v] \leq l[u]+w(u,v) \) => longer path => \( P_j \) does nothing
  - If \( sp[v] > l[u]+w(u,v) \) => \( P_j \) must insert \( v \) back into \( Q_j \) with \( l[v] = l[u]+w(u,v) \) and disregards the value \( sp[v] \)
- The algorithm terminates when all the queues are empty.
Johnson’s Algorithm: Distributed Memory Formulation

The wave of activity in the priority queues for a grid graph