Graph Algorithms

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text "Introduction to Parallel Computing", Addison Wesley, 2003
Topic Overview

- Definitions and Representation
- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Connected Components
- Algorithms for Sparse Graphs
Definitions and Representation

• An **undirected graph** $G$ is a pair $(V,E)$, where $V$ is a finite set of points called *vertices* and $E$ is a finite set of *edges*.
• An edge $e \in E$ is an unordered pair $(u,v)$, where $u,v \in V$.
• In a **directed graph**, the edge $e$ is an ordered pair $(u,v)$. An edge $(u,v)$ is *incident from* vertex $u$ and is *incident to* vertex $v$.
• A **path** from a vertex $v$ to a vertex $u$ is a sequence $<v_0,v_1,v_2,...,v_k>$ of vertices where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, ..., k-1$.
• The **length of a path** is defined as the number of edges in the path.
Definitions and Representation

a) An undirected graph and (b) a directed graph.
Definitions and Representation

- An undirected graph is **connected** if every pair of vertices is connected by a path.
- A **forest** is an acyclic graph, and a **tree** is a connected acyclic graph.
- A graph that has weights associated with each edge is called a **weighted graph**.
Definitions and Representation

• Graphs can be represented by their **adjacency matrix** or an **edge** (or vertex) **list**.

• Adjacency matrices have a value \( a_{i,j} = 1 \) if nodes \( i \) and \( j \) share an edge; 0 otherwise. In case of a weighted graph, \( a_{i,j} = w_{i,j} \), the weight of the edge.

• The **adjacency list** representation of a graph \( G = (V,E) \) consists of an array \( \text{Adj}[1..|V|] \) of lists. Each list \( \text{Adj}[v] \) is a list of all vertices adjacent to \( v \).

• For a graph with \( n \) nodes, adjacency matrices take \( \Theta(n^2) \) space and adjacency list takes \( \Theta(|E|) \) space.
Definitions and Representation

An undirected graph and its **adjacency matrix** representation.

An undirected graph and its **adjacency list** representation.
Minimum Spanning Tree

• A *spanning tree* of an undirected graph $G$ is a subgraph of $G$ that is a tree containing all the vertices of $G$.
• In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
• A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.
Minimum Spanning Tree

An undirected graph and its minimum spanning tree.
Minimum Spanning Tree: Prim's Algorithm

• **Prim's algorithm** for finding an MST is a **greedy algorithm**.

• **Start by selecting an arbitrary vertex**, include it into the current MST.

• **Grow the current MST** by inserting into it the vertex **closest** to one of the vertices already in current MST.
Prim's minimum spanning tree algorithm.
**Minimum Spanning Tree: Prim's Algorithm**

1. \textbf{procedure} PRIM\_MST(V, E, w, r)  
2. \hspace{1em} begin  
3. \hspace{2em} $V_T := \{r\}$;  
4. \hspace{2em} $d[r] := 0$;  
5. \hspace{2em} \textbf{for} all $v \in (V - V_T)$ \textbf{do}  
6. \hspace{3em} \textbf{if} edge $(r, v)$ exists \textbf{set} $d[v] := w(r, v)$;  
7. \hspace{3em} \textbf{else} \textbf{set} $d[v] := \infty$;  
8. \hspace{2em} \textbf{while} $V_T \neq V$ \textbf{do}  
9. \hspace{3em} \textbf{begin}  
10. \hspace{4em} find a vertex $u$ such that $d[u] := \min\{d[v] | v \in (V - V_T)\}$;  
11. \hspace{4em} $V_T := V_T \cup \{u\}$;  
12. \hspace{4em} \textbf{for} all $v \in (V - V_T)$ \textbf{do}  
13. \hspace{5em} $d[v] := \min\{d[v], w(u, v)\}$;  
14. \hspace{4em} \textbf{endwhile}  
15. \hspace{2em} \textbf{end} PRIM\_MST

Prim's sequential minimum spanning tree algorithm.
Prim's Algorithm: Parallel Formulation

- The algorithm works in \textit{n outer iterations} - it is hard to execute these iterations concurrently.
- The inner loop is relatively easy to parallelize. Let \textit{p be the number of processes}, and let \textit{n be the number of vertices}.
- The \textit{adjacency matrix is partitioned in a 1-D block fashion}, with distance vector \textit{d} partitioned accordingly.
- In each step, a \textit{processor selects the locally closest node}, followed by a \textit{global reduction to select globally closest node}.
- This node is inserted into MST, and \textit{the choice broadcast to all processors}.
- Each \textit{processor updates} its part of the \textit{d} vector locally.
Prim's Algorithm: Parallel Formulation

The partitioning of the distance array $d$ and the adjacency matrix $A$ among $p$ processes.
Prim's Algorithm: Parallel Formulation

- The cost to **select the minimum entry** is $O(n/p + \log p)$.
- The cost of a **broadcast** is $O(\log p)$.
- The cost of **local updation** of the $d$ vector is $O(n/p)$.

- The **parallel time per iteration** is $O(n/p + \log p)$.
- The **total parallel time** is given by $O(n^2/p + n \log p)$.

- The corresponding **isoefficiency** is $O(p^2 \log^2 p)$. 
**Single-Source Shortest Paths**

- For a weighted graph $G = (V,E,w)$, the *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in $V$.
- Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.
- It grows this set based on the node closest to source using one of the nodes in the current shortest path set.
Single-Source Shortest Paths: Dijkstra's Algorithm

```
procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
begin
    V_T := {s};
    for all v ∈ (V − V_T) do
        if (s, v) exists set l[v] := w(s, v);
        else set l[v] := ∞;
    while V_T ≠ V do
        begin
            find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
            V_T := V_T ∪ {u};
            for all v ∈ (V − V_T) do
                l[v] := min{l[v], l[u] + w(u, v)};
        endwhile
    end DIJKSTRA_SINGLE_SOURCE_SP
```

Dijkstra's sequential single-source shortest paths algorithm.
Dijkstra's Algorithm: Parallel Formulation

• Very similar to the parallel formulation of Prim's algorithm for minimum spanning trees.
• The weighted adjacency matrix is partitioned using the 1-D block mapping.
• Each process selects, locally, the node closest to the source, followed by a global reduction to select next node.
• The node is broadcast to all processors and the l-vector updated.
• The parallel performance of Dijkstra's algorithm is identical to that of Prim's algorithm.
All-Pairs Shortest Paths

• Given a weighted graph $G(V,E,w)$, the *all-pairs shortest paths* problem is to find the *shortest paths between all pairs* of vertices $v_i, v_j \in V$.

• A number of algorithms are known for solving this problem.
All-Pairs Shortest Paths: Matrix-Multiplication Based Algorithm

• Consider the multiplication of the weighted adjacency matrix with itself - except, in this case, we replace the multiplication operation in matrix multiplication by addition, and the addition operation by minimization.

• Notice that the product of weighted adjacency matrix with itself returns a matrix that contains shortest paths of length 2 between any pair of nodes.

• It follows from this argument that $A^n$ contains all shortest paths.
Matrix-Multiplication Based Algorithm

\[ A^1 = \begin{pmatrix}
0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty & \infty \\
\infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & 0 & \infty & 2 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty
\end{pmatrix} \quad A^2 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty & \infty \\
\infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty
\end{pmatrix} \]

\[ A^4 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\
\infty & 0 & \infty & 4 & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty
\end{pmatrix} \quad A^8 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\
\infty & 0 & \infty & 4 & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty
\end{pmatrix} \]
Matrix-Multiplication Based Algorithm

• $A^n$ is computed by **doubling powers** - i.e., as $A, A^2, A^4, A^8,$ and so on.

• We need **log n matrix multiplications**, each taking time $O(n^3)$.

• The **serial complexity** of this procedure is $O(n^3 \log n)$.

• This algorithm is **not optimal**, since the best known algorithms have complexity $O(n^3)$. 
Matrix-Multiplication Based Algorithm: Parallel Formulation

- Each of the $\log n$ matrix multiplications can be performed in parallel.
- We can use $n^3/\log n$ processors to compute each matrix-matrix product in time $\log n$.
- The entire process takes $O(\log^2 n)$ time.
Dijkstra's Algorithm

- Execute $n$ instances of the single-source shortest path problem, one for each of the $n$ source vertices.
- Complexity is $O(n^3)$. 
Dijkstra's Algorithm: Parallel Formulation

• **Two parallelization strategies** - execute each of the $n$ shortest path problems on a different processor (**source partitioned**), or use a parallel formulation of the shortest path problem to increase concurrency (**source parallel**).
Dijkstra's Algorithm: Source Partitioned Formulation

- Use *n processors*, each processor $P_i$ finds the shortest paths from vertex $v_i$ to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.
- It requires **no interprocess communication** (provided that the adjacency matrix is replicated at all processes).
- The **parallel run time** of this formulation is: $\Theta(n^2)$.
- While the algorithm is cost optimal, it can only use *n processors*. Therefore, the isoefficiency due to concurrency is $p^3$. 
Dijkstra's Algorithm: Source Parallel Formulation

- In this case, each of the shortest path problems is further executed in parallel. We can therefore use up to \( n^2 \) processors.
- Given \( p \) processors (\( p > n \)), each single source shortest path problem is executed by \( p/n \) processors.
- Using previous results, this takes time:

\[
T_P = \Theta \left( \frac{n^3}{p} \right) + \Theta(n \log p).
\]

- For cost optimality, we have \( p = O(n^2/\log n) \) and the isoefficiency is \( \Theta((p \log p)^{1.5}) \).
Floyd's Algorithm

- For any pair of vertices $v_i, v_j \subseteq V$, consider all paths from $v_i$ to $v_j$ whose intermediate vertices belong to the set $\{v_1, v_2, \ldots, v_k\}$. Let $p_i^{(k)}$ (of weight $d_i^{(k)}$) be the minimum-weight path among them.

- If vertex $v_k$ is not in the shortest path from $v_i$ to $v_j$, then $p_i^{(k)}$ is the same as $p_i^{(k-1)}$.

- If $v_k$ is in $p_i^{(k)}$, then we can break $p_i^{(k)}$ into two paths - one from $v_i$ to $v_k$ and one from $v_k$ to $v_j$. Each of these paths uses vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$. 

Floyd's Algorithm

From our observations, the following recurrence relation follows:

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

This equation must be computed for each pair of nodes and for \( k = [1, n] \). The serial complexity is \( O(n^3) \).
Floyd's Algorithm

1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3. \[ D^{(0)} = A; \]
4. \[ \text{for } k := 1 \text{ to } n \text{ do} \]
5. \[ \quad \text{for } i := 1 \text{ to } n \text{ do} \]
6. \[ \quad \quad \text{for } j := 1 \text{ to } n \text{ do} \]
7. \[ d_{i,j}^{(k)} := \min \left( d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right); \]
8. end FLOYD_ALL_PAIRS_SP

Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph \( G = (V,E) \) with adjacency matrix \( A \).
Floyd's Algorithm: Parallel Formulation
Using 2-D Block Mapping

- Matrix $D^{(k)}$ is divided into $p$ blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$.

- Each processor updates its part of the matrix during each iteration.

- To compute $d_{l, (k-1)}$ processor $P_{i,j}$ must get $d_{l, (k-1)}$ and $d_{k, (r-1)}$.

- In general, during the $k^{th}$ iteration, each of the $\sqrt{p}$ processes containing part of the $k^{th}$ row send it to the $\sqrt{p} - 1$ processes in the same column.

- Similarly, each of the $\sqrt{p}$ processes containing part of the $k^{th}$ column sends it to the $\sqrt{p} - 1$ processes in the same row.
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

(a) Matrix $D^{(k)}$ distributed by 2-D block mapping into $\sqrt{p} \times \sqrt{p}$ subblocks, and (b) the subblock of $D^{(k)}$ assigned to process $P_{i,j}$. 
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

(a) Communication patterns used in the 2-D block mapping. When computing $d_{i,j}^{(k)}$, information must be sent to the highlighted process from two other processes along the same row and column. (b) The row and column of $\sqrt{p}$ processes that contain the $k^{th}$ row and column send them along process columns and rows.
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

1. procedure FLOYD_2DBLOCK(D^{(0)})
2. begin
3. for k := 1 to n do
4. begin
5. each process P_{i,j} that has a segment of the k^{th} row of D^{(k-1)};
   broadcasts it to the P_{*,j} processes;
6. each process P_{i,j} that has a segment of the k^{th} column of D^{(k-1)};
   broadcasts it to the P_{i,*} processes;
7. each process waits to receive the needed segments;
8. each process P_{i,j} computes its part of the D^{(k)} matrix;
9. end
10. end FLOYD_2DBLOCK

Floyd's parallel formulation using the 2-D block mapping. P_{*,j} denotes all the processes in the j^{th} column, and P_{i,*} denotes all the processes in the i^{th} row. The matrix D^{(0)} is the adjacency matrix.
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- During each iteration of the algorithm, the $k^{th}$ row and $k^{th}$ column of processors perform a one-to-all broadcast along their rows/columns.
- The size of this broadcast is $n\sqrt[p]{p}$ elements, taking time $\Theta((n \log p)/\sqrt{p})$.
- The synchronization step takes time $\Theta(\log p)$.
- The computation time is $\Theta(n^2/p)$.
- The parallel run time of the 2-D block mapping formulation of Floyd's algorithm is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta\left(\frac{n^2}{\sqrt{p}\log p}\right).$$
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

• The above formulation can use $O(n^2 / \log^2 n)$ processors cost-optimally.
• The isoefficiency of this formulation is $\Theta(p^{1.5} \log^3 p)$.
• This algorithm can be further improved by relaxing the strict synchronization after each iteration.
Floyd's Algorithm: Speeding Things Up by Pipelining

- The **synchronization step** in parallel Floyd's algorithm can be removed without affecting the correctness of the algorithm.
- A process starts working on the $k^{th}$ iteration as soon as it has computed the $(k-1)^{th}$ iteration and has the relevant parts of the $D^{(k-1)}$ matrix.
Floyd's Algorithm: Speeding Things Up by Pipelining

- The overall **parallel run time** of this formulation is

\[ T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n). \]

- The pipelined formulation of Floyd's algorithm **uses up to** \( O(n^2) \) processes **efficiently**.
- The corresponding **isoefficiency** is \( \Theta(p^{1.5}) \).
All-pairs Shortest Path: Comparison

- The performance and scalability of the all-pairs shortest paths algorithms on various architectures with bisection bandwidth. Similar run times apply to all cube architectures, provided that processes are properly mapped to the underlying processors.

<table>
<thead>
<tr>
<th>Maximum Number of Processes for $E = \Theta(1)$</th>
<th>Corresponding Parallel Run Time</th>
<th>Isoefficiency Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra source-partitioned</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(p^3)$</td>
</tr>
<tr>
<td>Dijkstra source-parallel</td>
<td>$\Theta(n^2 / \log n)$</td>
<td>$\Theta((p \log p)^{1.5})$</td>
</tr>
<tr>
<td>Floyd 2-D block</td>
<td>$\Theta(n^2 / \log^2 n)$</td>
<td>$\Theta(p^{1.5} \log^3 p)$</td>
</tr>
<tr>
<td>Floyd pipelined 2-D block</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(p^{1.5})$</td>
</tr>
</tbody>
</table>
Connected Components

- The **connected components** of an **undirected graph** are the equivalence classes of vertices under the ``is reachable from'' relation.

A graph with three connected components: \{1,2,3,4\}, \{5,6,7\}, and \{8,9\}. 
Connected Components: Depth-First Search Based Algorithm

- Perform **DFS on the graph to get a forest** - arc tree in the forest corresponds to a separate connected component.

Part (b) is a **depth-first forest** obtained from depth-first traversal of the **graph in part (a)**. Each of these trees is a connected component of the graph in part (a).
Connected Components: Parallel Formulation

- Partition the graph across processors and run independent connected component algorithms on each processor. At this point, we have $p$ spanning forests.
- In the second step, spanning forests are merged pairwise until only one spanning forest remains.
Computing connected components in parallel. The adjacency matrix of the graph $G$ in (a) is partitioned into two parts (b). Each process gets a subgraph of $G$ ((c) and (e)). Each process then computes the spanning forest of the subgraph ((d) and (f)). Finally, the two spanning trees are merged to form the solution.
Connected Components: Parallel Formulation

• To merge pairs of spanning forests efficiently, the algorithm uses disjoint sets of edges.

• We define the following operations on the disjoint sets:
  
  • **find(x)**
    – returns a pointer to the representative element of the set containing \( x \). Each set has its own unique representative.
  
  • **union(x, y)**
    – unites the sets containing the elements \( x \) and \( y \). The two sets are assumed to be disjoint prior to the operation.
Connected Components: Parallel Formulation

- For merging forest $A$ into forest $B$, for each edge $(u, v)$ of $A$, a find operation is performed to determine if the vertices are in the same tree of $B$.
- If not, then the two trees (sets) of $B$ containing $u$ and $v$ are united by a union operation.
- Otherwise, no union operation is necessary.
- Hence, merging $A$ and $B$ requires at most $2(n-1)$ find operations and $(n-1)$ union operations.
Connected Components: Parallel 1-D Block Mapping

- The $n \times n$ adjacency matrix is partitioned into $p$ blocks (1-D).
- Each processor can compute its local spanning forest in time $\Theta(n^2/p)$.
- Merging is done by embedding a logical tree into the topology. There are $\log p$ merging stages, and each takes time $\Theta(n)$. Thus, the cost due to merging is $\Theta(n \log p)$.
- During each merging stage, spanning forests are sent between nearest neighbors. Recall that $\Theta(n)$ edges of the spanning forest are transmitted.
Connected Components: Parallel 1-D Block Mapping

- The parallel run time of the connected-component algorithm is

\[ T_p = \Theta \left( \frac{n^2}{p} \right) + \Theta(n \log p). \]

- For a cost-optimal formulation \( p = O(n / \log n) \). The corresponding isoefficiency is \( \Theta(p^2 \log^2 p) \).
A graph $G = (V,E)$ is sparse if $|E|$ is much smaller than $|V|^2$.

Examples of sparse graphs: (a) a linear graph, in which each vertex has two incident edges; (b) a grid graph, in which each vertex has four incident vertices; and (c) a random sparse graph.
Algorithms for Sparse Graphs

• Dense algorithms can be improved significantly if we make use of the sparseness. For example, the run time of Prim's minimum spanning tree algorithm can be reduced from $\Theta(n^2)$ to $\Theta(|E| \log n)$.

• Sparse algorithms use adjacency list instead of an adjacency matrix.

• Partitioning adjacency lists is more difficult for sparse graphs - do we balance number of vertices or edges?

• Parallel algorithms typically make use of graph structure or degree information for performance.
Finding a Maximal Independent Set

• A set of vertices $I \subset V$ is called *independent* if no pair of vertices in $I$ is connected via an edge in $G$. An independent set is called *maximal* if by including any other vertex not in $I$, the independence property is violated.

Examples of independent and maximal independent sets.

\begin{itemize}
  \item \{a, d, i, h\} is an independent set
  \item \{a, c, j, f, g\} is a maximal independent set
  \item \{a, d, h, f\} is a maximal independent set
\end{itemize}
Finding a Maximal Independent Set (MIS)

- Simple algorithms start by MIS $I$ to be empty, and assigning all vertices to a candidate set $C$.
- Vertex $v$ from $C$ is moved into $I$ and all vertices adjacent to $v$ are removed from $C$.
- This process is repeated until $C$ is empty.
- This process is inherently serial!
Finding a Maximal Independent Set (MIS)

- Parallel MIS algorithms use randomization to gain concurrency (Luby's algorithm for graph coloring).
- Initially, each node is in the candidate set $C$. Each node generates a (unique) random number and communicates it to its neighbors.
- If a node's number is smallest out of all its neighbors, it joins set $I$. All of its neighbors are removed from $C$.
- This process continues until $C$ is empty.
- On average, this algorithm converges after $O(\log |V|)$ such steps.
Finding a Maximal Independent Set (MIS)

The different augmentation steps of Luby's randomized maximal independent set algorithm. The numbers inside each vertex correspond to the random number assigned to the vertex.

(a) After the 1st random number assignment

(b) After the 2nd random number assignment

(c) Final maximal independent set
Finding a Maximal Independent Set (MIS): Parallel Formulation

• We use **three arrays**, each of length \( n \). Array \( I \), which stores nodes in MIS, \( C \), which stores the candidate set, and \( R \), the random numbers.

• Partition \( C \) across \( p \) processors. **Each processor generates the corresponding values in the** \( R \) **array, and from this, computes which candidate vertices can enter MIS.**

• The **\( C \) array is updated by deleting all the neighbors of vertices that entered \( MIS \).**

• The performance of this algorithm is dependent on the structure of the graph.