Course Name: Analysis and Design of Algorithms

Topics to be covered

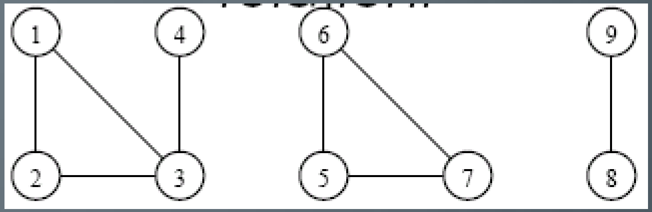
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

Transitive Closure

- If $G = (V,E)$ is a graph, then the *transitive closure* of G is defined as the graph $G^* = (V,E^*)$, where $E^* = \{(v_i,v_j) \mid \text{there is a path from } v_i \text{ to } v_j \text{ in } G\}$
- The *connectivity matrix* of G is a matrix $A^* = (a_{i,j}^*)$ such that $a_{i,j}^* = 1$ if there is a path from v_i to v_j or $i = j$, and $a_{i,j}^* = \infty$ otherwise.
- To compute A^* we assign a weight of 1 to each edge of E and use any of the all-pairs shortest paths algorithms on this weighted graph.

Connected Components

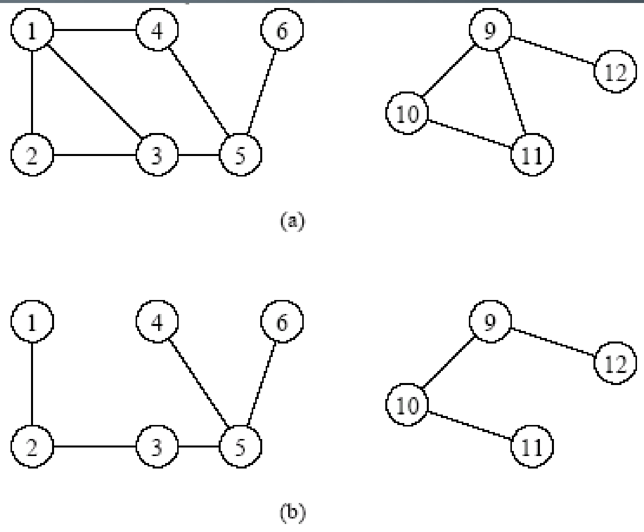
- The connected components of an undirected graph are the equivalence classes of vertices under the ``is reachable from'' relation.



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 - eac tree in the forest corresponds to a separate connected component.

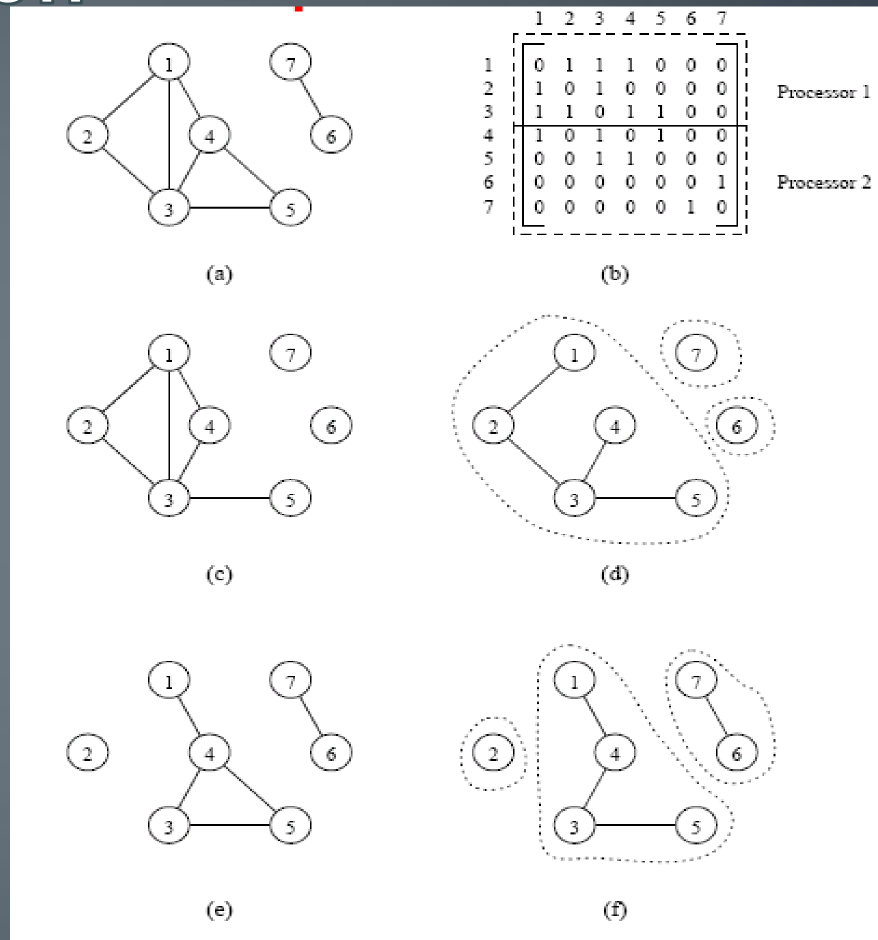


Part (b) is a depth-first forest obtained from depth-first traversal of the graph in part (a). Each of these trees is a connected component of the graph in part (a).

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.

Connected Components: Parallel Formulation



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 x n adjacency matrix is partitioned into p blocks.
- 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

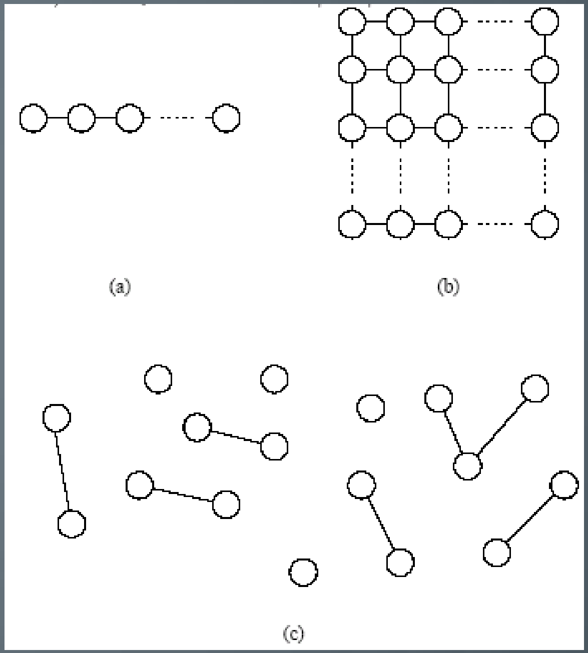
- The parallel run time of the connected-component algorithm is

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{local computation}} + \overbrace{\Theta(n \log p)}^{\text{forest merging}}.$$

- For a cost-optimal formulation *$p = O(n / \log n)$*. The corresponding isoefficiency is *$\Theta(p^2 \log^2 p)$*.

Algorithms for Sparse Graphs

- A graph $G = (V,E)$ is sparse if $|E|$ is much smaller than $|V|^2$.

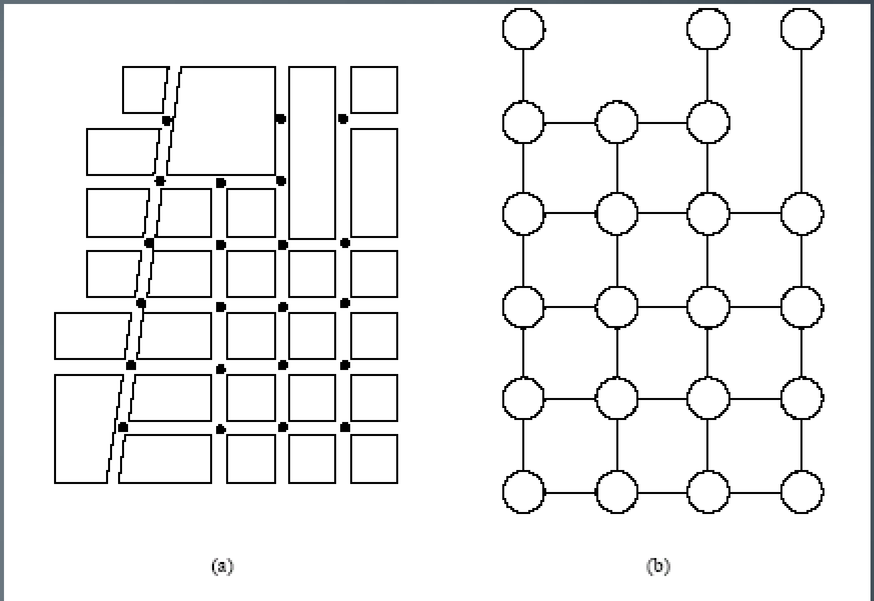


Examples of sparse graphs: (a) a linear graph, in which each vertex has two incident edges; (b) a grid graph, in which each vertex has four incident vertices; and (c) a random sparse graph.

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.

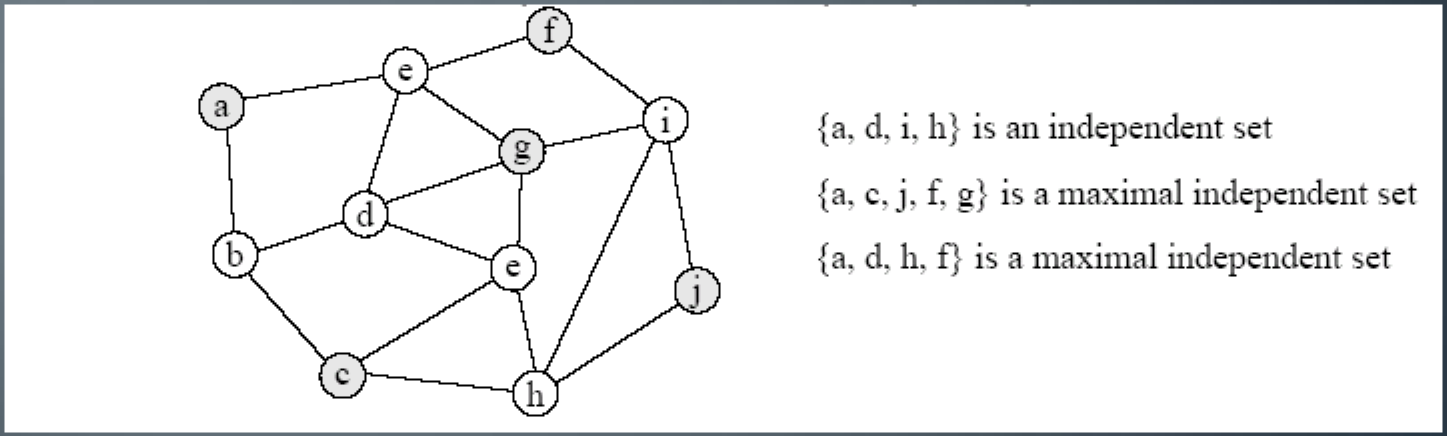
Algorithms for Sparse Graphs



A street map (a) can be represented by a graph (b). In the graph shown in (b), each street intersection is a vertex and each edge is a street segment. The vertices of (b) are the intersections of (a) marked by dots.

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.

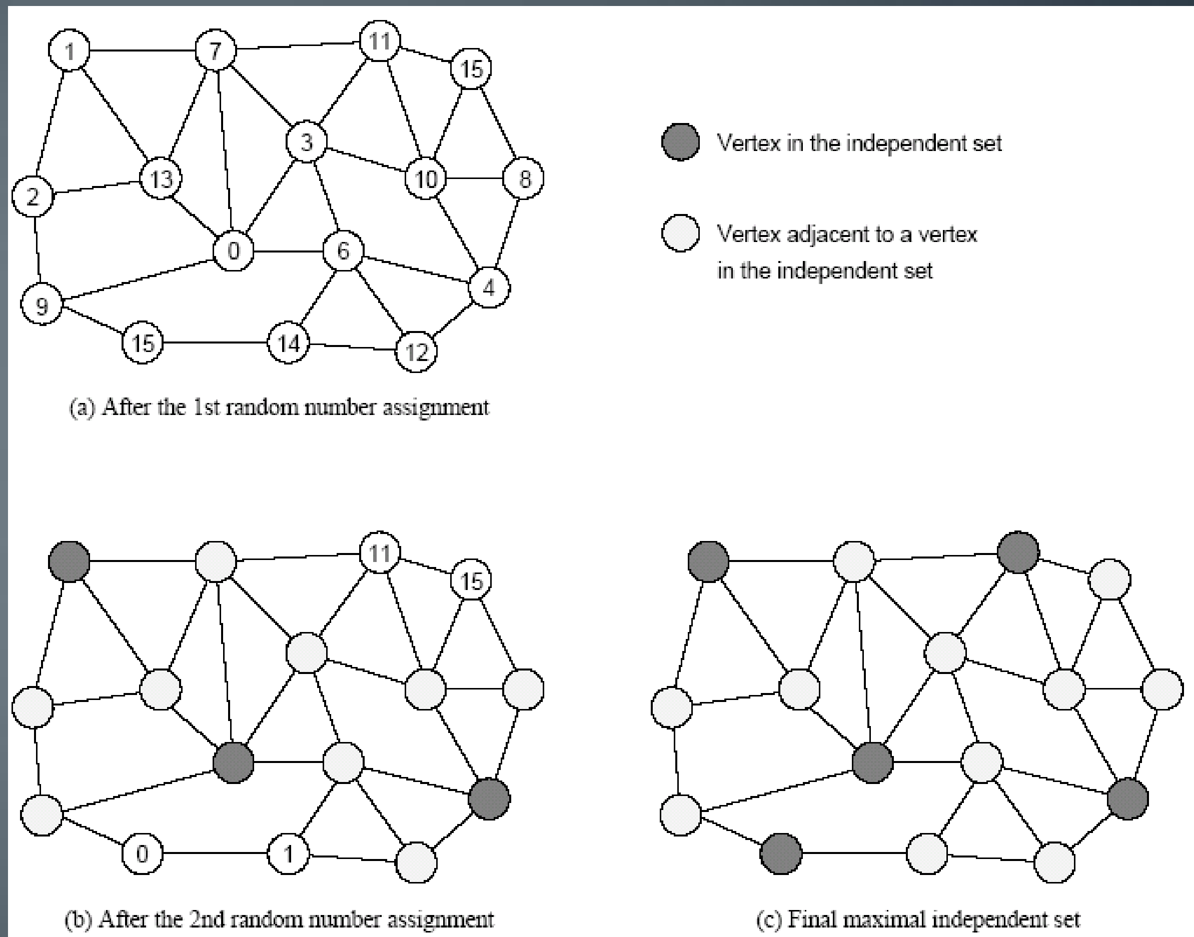
Finding a Maximal Independent Set (MIS)

- 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 randimization 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 nodes number exceeds that 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)



(a) After the 1st random number assignment

(b) After the 2nd random number assignment

(c) Final maximal independent set

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.

Finding a Maximal Independent Set (MIS): Parallel Formulation

- We use three arrays, each of length n - 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.

Single-Source Shortest Paths

- Dijkstra's algorithm, modified to handle sparse graphs is called Johnson's algorithm.
- The modification accounts for the fact that the minimization step in Dijkstra's algorithm needs to be performed only for those nodes adjacent to the previously selected nodes.
- Johnson's algorithm uses a priority queue Q to store the value $l[v]$ for each vertex $v \in (V - V_T)$.

Single-Source Shortest Paths: Johnson's Algorithm

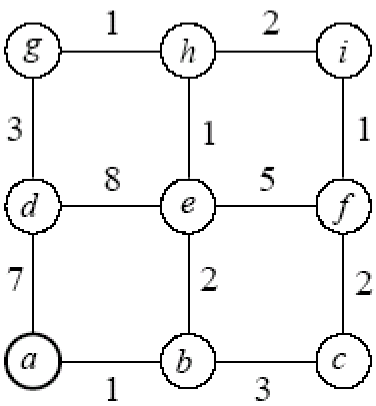
```
procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
begin
    Q := V;
    for all v ∈ Q do
        l[v] := ∞;
    l[s] := 0;
    while Q ≠ ∅ do
    begin
        u := extract_min(Q);
        for each v ∈ Adj[u] do
            if v ∈ Q and l[u] + w(u, v) < l[v] then
                l[v] := l[u] + w(u, v);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

Johnson's sequential single-source shortest paths algorithm.

Single-Source Shortest Paths: Parallel Johnson's Algorithm

- Maintaining strict order of Johnson's algorithm generally leads to a very restrictive class of parallel algorithms.
- We need to allow exploration of multiple nodes concurrently. This is done by simultaneously extracting p nodes from the priority queue, updating the neighbors' cost, and augmenting the shortest path.
- If an error is made, it can be discovered (as a shorter path) and the node can be reinserted with this shorter path.

Single-Source Shortest Paths: Parallel Johnson's Algorithm



	Priority Queue	a	b	c	d	e	f	g	h	i
(1)	*b:1, d:7, c:inf, e:inf, f:inf, g:inf, h:inf, i:inf*	0	1	∞	7	∞	∞	∞	∞	∞
(2)	*e:3, c:4, g:10, f:inf, h:inf, i:inf*	0	1	4	7	3	∞	10	∞	∞
(3)	*h:4, f:6, i:inf*	0	1	4	7	3	6	10	4	∞
(4)	*g:5, i:6*	0	1	4	7	3	6	5	4	6

Array l[]

An example of the modified Johnson's algorithm for processing unsafe vertices concurrently.

Single-Source Shortest Paths: Parallel Johnson's Algorithm

- Even if we can extract and process multiple nodes from the queue, the queue itself is a major bottleneck.
- For this reason, we use multiple queues, one for each processor. Each processor builds its priority queue only using its own vertices.
- When process P_i extracts the vertex $u \in V_i$, it sends a message to processes that store vertices adjacent to u.
- Process P_j, upon receiving this message, sets the value of $l[v]$ stored in its priority queue to $min\{l[v], l[u] + w(u,v)\}$.

Single-Source Shortest Paths: Parallel Johnson's Algorithm

- If a shorter path has been discovered to node *v*, it is reinserted back into the local priority queue.
- The algorithm terminates only when all the queues become empty.
- A number of node paritioning schemes can be used to exploit graph structure for performance.