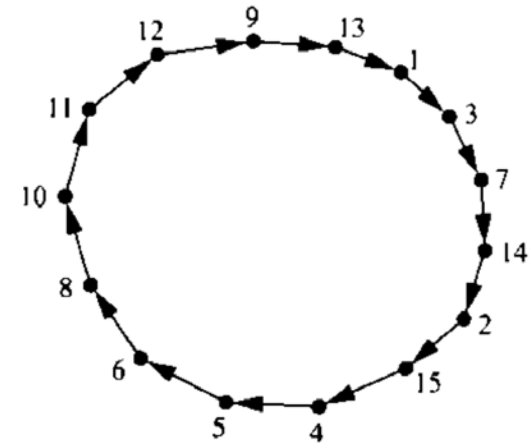
PRAM 2
Graph algorithms

CS121 Parallel Computing

Spring 2019

Coloring a cycle



- Given a graph G=(V,E), a k-coloring of G is a mapping $c: V \to \{0,1,\dots,k-1\}$ s.t. $c(i) \neq c(j)$ whenever $(i,j) \in E$.
- We give a super fast algorithm for 3-coloring a (directed) cycle of n nodes.
  - If n is odd, any coloring uses at least 3 colors.
  - Coloring the cycle is a form of symmetry breaking.
  - For any node v, let S(v) be the node after v.
- The main subroutine is the following.
  - Initially, color every node by its node ID.
  - Consider the binary representation of c(v) for a node v.
  - Let k be the least significant digit in which c(v) and c(S(v)) differ.
  - Set $c'(v)=2k+c(v)_k$, where $c(v)_k$ is the k'th digit of c(v).
- Claim If c is a valid coloring, then so is c'.
- Proof Since c is a valid coloring, then $c(v) \neq c\big(S(v)\big)$, so k exists.
  - Suppose $c'(v) = c'(u)$, for some v and $u = S(v)$.
  - Then $c'(v) = 2k + c(v)_k$ and $c'(u) = 2l + c(u)_l$ for some k and l.
  - Since $c'(v) = c'(u)$, then $k = l$, because $c(v)_k, c(u)_l < 2$.
  - But then $c(v)_k = c(u)_k$, contradicting the definition of k.

v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

Coloring a cycle

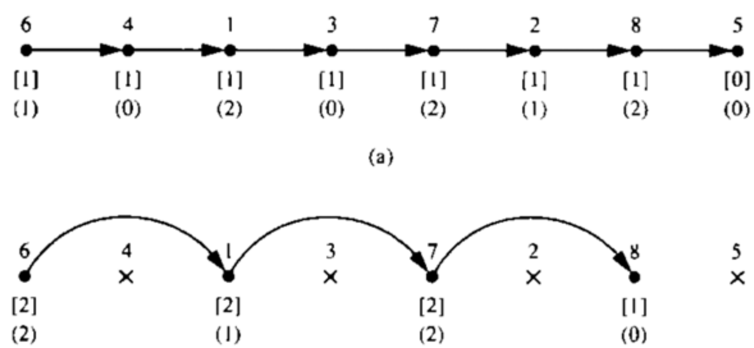
- To analyze the time complexity, suppose in some round the max number of bits to represent any color is t.
- Then the max number of bits to represent any color in the next round is $\lceil \log t \rceil + 1$, because any color in the next round is $\leq 2t + 1$.
  - So the number of bits used to represent a color decreases from t to $\lceil \log t \rceil + 1$ in each round.
- Let $\log^{(i)} x = \log(\log^{(i-1)} x)$, i.e. we apply the log function i times to x.
- Let $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$ be the number of times we have to take log's until a value becomes ≤ 1.
  - $\log^* x$ is incredibly small. In fact, $\log^* x \leq 6$ for all $x \leq 2^{65536}$!

Coloring a cycle

- Since the number of bits to represent a color in the first round is $\log n$, then in $O(\log^* n)$ rounds, we can represent any color using O(1) bits.
  - In fact, we can apply the subroutine until we use 6 colors in a round.
  - With 6 colors, need 3 bits to represent a color. So in the next round, colors are between 0 and 2*2+1=5, and we again use up to 6 colors.
- To decrease the number of colors from 6 to 3, we run 3 more rounds.
  - In round i, take any node colored using color i+2 and color it using the min possible color in {0,1,2}, i.e. the min color not used by its neighbors.
- In total, we 3-color the ring in $O(\log^* n)$ rounds, using $O(n \log^* n)$ work.
- The algorithm can be modified to produce a 3-coloring in $O(\log n)$ time and $O(n)$ work.

Independent set on line

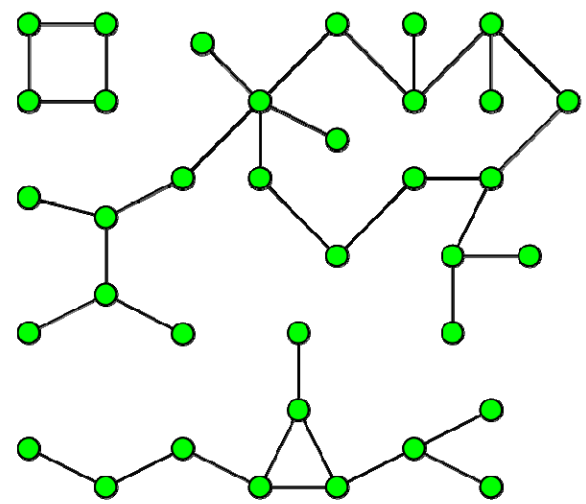
- Thm Given a k coloring of a line graph with n nodes, we can compute an independent set of size $\Omega(n/k)$ in $O(1)$ time.
- Proof Every node has a color from 1 to k.
  - Take the nodes whose colors are local minima as the indep. set S.
    - No two nodes in S are neighbors.
    - S can be computed in $O(1)$ time.
  - Consider two consecutive nodes $u, v \in S$.
  - Since u, v are local minima and consecutive, the colors between u, v first increase, then decrease.
  - Thus, there are at most $2k-3$ nodes between u and v.
  - So there are $\leq 2k-3$ nodes between any two consecutive nodes in S, and so $|S| \geq \frac{n}{2k-3}$.
- Thus, compute independent set of size $\Omega(n)$ on line with n nodes in $O(\log n)$ time by computing a 3 coloring of the line in $O(\log n)$ time, then computing an independent set of size $\Omega(n/3) = \Omega(n)$ in $O(1)$ time.



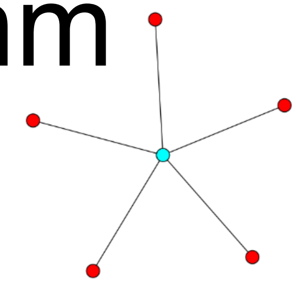
- The colors of the nodes are shown in parentheses.
- Nodes 4, 3, 2, 5 are local minima.

Connected components

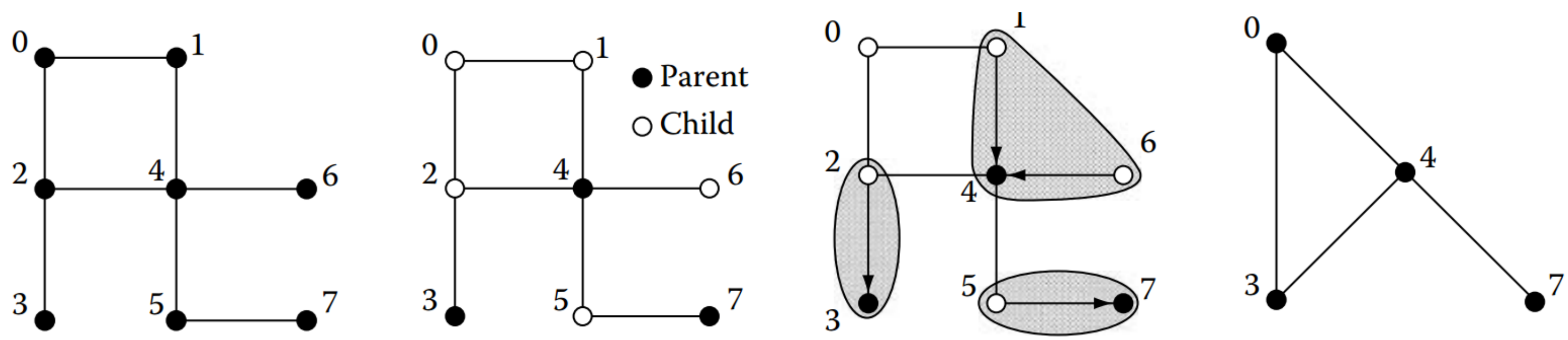
- Given an undirected graph, partition it into maximal sets of nodes that are connected to each other.
- Can be solved sequentially in O(m) time using BFS / DFS.
  - m is number of edges, n is number of vertices.
- However, no efficient BFS / DFS PRAM algorithms known.
- Instead, use graph contractions.
  - In each phase, merge (contract) a set of connected nodes into a supernode.
  - Form a contracted graph on the supernodes, then apply algorithm recursively.
  - Eventually each connected component is contracted to one node.
  - Many different algorithms, depending on which nodes they contract.



Randomized parallel algorithm



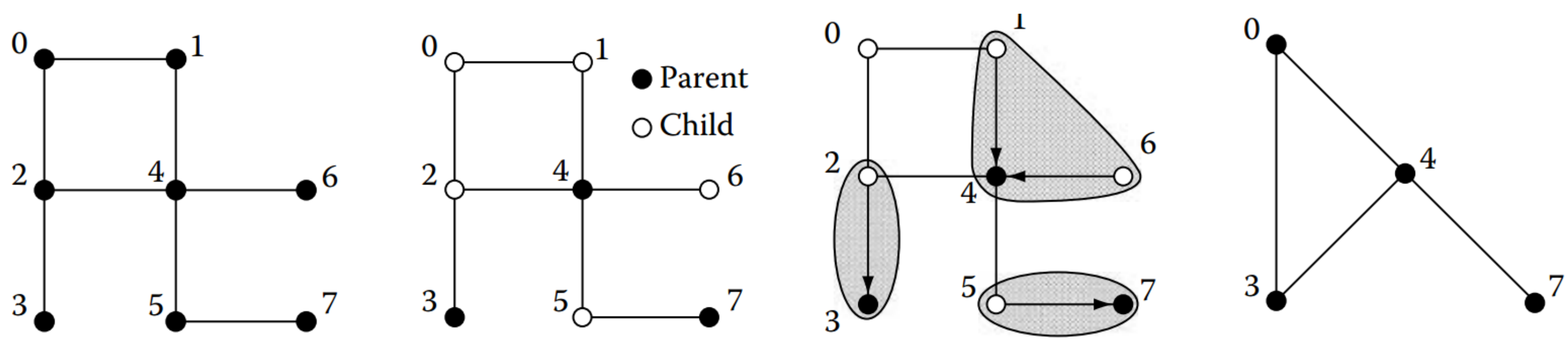
- Break graph into star graphs and contract each star.
- In each phase, do the following steps in parallel.
  - Every node flips a coin and chooses to be a parent or child node.
  - Each child node points to a parent node it's connected to.
    - Now have a set of stars, with the parent nodes as the centers.
    - If child not connected to any parent node, it forms its own star.
  - Contract each star to its center, then apply algorithm recursively.
    - Label all nodes in star by the label of the parent.
    - Keep the edges between differently labeled nodes.
  - After recursion returns, each child with a parent again takes parent's label, which might have changed.



Source: Algorithms and Theory of Computation Handbook, Chapter 25

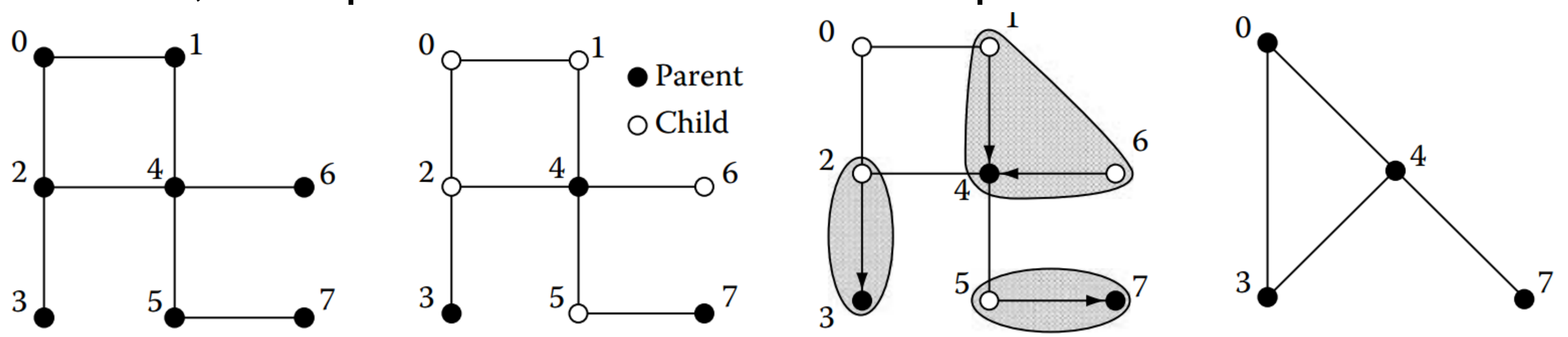
Implementation

- Use n+m processors, one processor for each vertex and edge.
  - Call these E and V procs, resp.
  - Each V proc has a label which can change over time.
  - Each E proc responsible for edge (u,v), where u, v are V procs.
  - V and E procs may become inactive over time.
- In each phase, each active V proc flips a coin to decide if it’s a child or parent proc.
- Each active E proc (u,v) checks if u is a child proc and v is a parent (or vice versa).
  - If so, it sets u’s label to v (or v’s label to u).
  - Another E proc (u,v’) could set u’s label to v’. In this case, either the v or v’ write succeeds.



Implementation

- Each parent V proc, or child V proc whose label didn't change (i.e. it had no parent), stays active.
  - Other V procs become inactive.
- Each active E proc (u,v) where u, v have different labels stays active.
  - Other E procs become inactive.
  - From now on, E will be responsible for V processes (u',v'), where u' and v' are the labels of u and v, resp.
- The active V and E procs run the algorithm recursively.
- After recursion returns, inactive E procs (u,v) (where u is the child) write v's label to u.
- At end, all V procs in a connected component have same label.

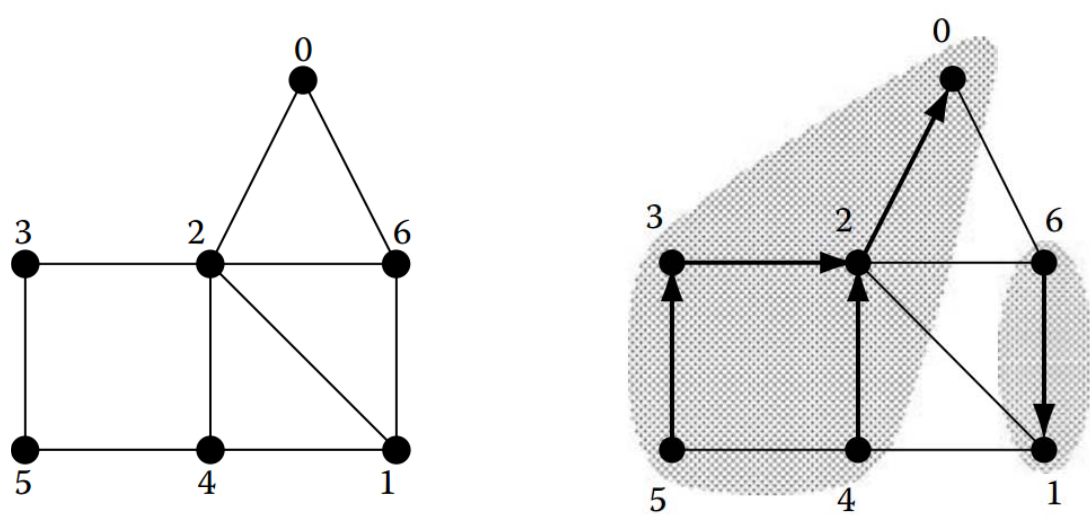


Complexity

- Key fact is that number of active V procs decreases by 1/4 fraction in expectation each phase.
  - A V proc becomes inactive if it's a child node and one its neighbors is a parent node.
  - The former probability is 1/2, and the latter is $\geq 1/2$.
  - Thus each V proc becomes inactive with probability $\geq 1/4$.
- With high probability, after O(log n) phases, there's only one V proc and recursion ends.
- Each phase takes O(1) time, and does O(m+n) work.
- Total time is O(log n), total work is O((m+n) log n).
  - This algorithm isn't work efficient.
  - There exist work efficient randomized CC algorithms.

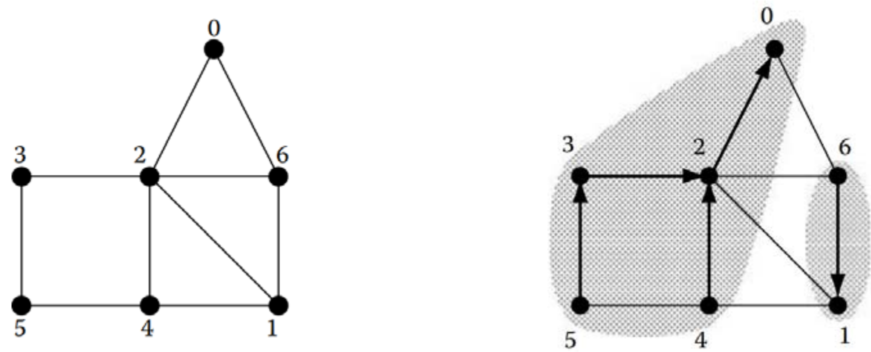
Deterministic parallel algorithm

- Again work in phases, with following parallel steps.
  - Each node points to a neighbor with lower ID.
  - This breaks graph into a directed forest.
  - Contract each forest to the lowest ID node using pointer jumping.
  - Recurse on contracted graph.



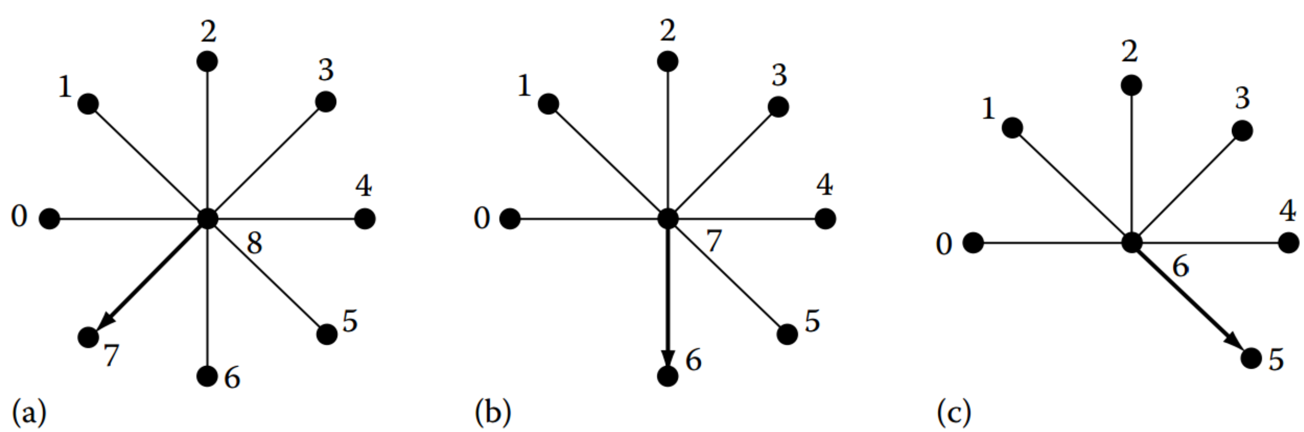
Implementation

- As before, use n+m procs, for V and E.
  - But now, V procs always active. E procs may become inactive.
- Each E proc (u,v) checks if u<v, and if so sets v's label to u.
  - Again, conflicts resolved arbitrarily.
- V procs then apply pointer jumping on the labels, taking the label of the proc it points to.
- Each active E proc (u,v) where u, v have different labels stays active. Other E procs become inactive.
  - From now on, E will be responsible for V processes (u',v'), where u' and v' are the labels of u and v, resp.
  - There may be multiple E procs responsible for same (u',v').
- The V procs and active E procs run algorithm recursively.
  - Note that in recursive call, all V procs apply pointer jumping.



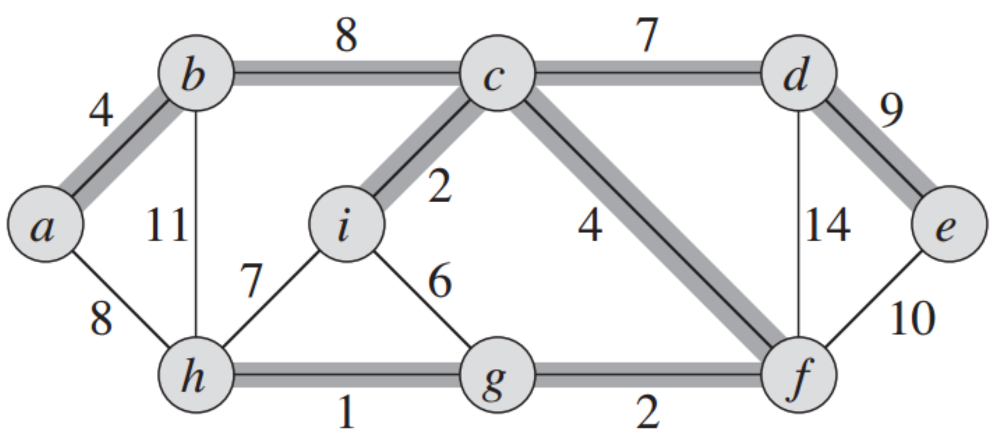
Complexity

- The basic algorithm may take O(n) time on the graph below.
- But notice that if we made nodes point to higher neighbors, the graph would be solved in O(1) time.
- In each phase, if we consider either having nodes point to smaller neighbors, or pointing to higher neighbors, in one case $\geq n/2$ nodes point to another node.
- Thus the algorithm finishes in O(log n) phases.
- Each phase does pointer jumping, which takes O(log n) time and O(n log n) work.
- Total time is $O(\log^2 n)$, and work is O((m+n log n) log n).



Minimum spanning tree

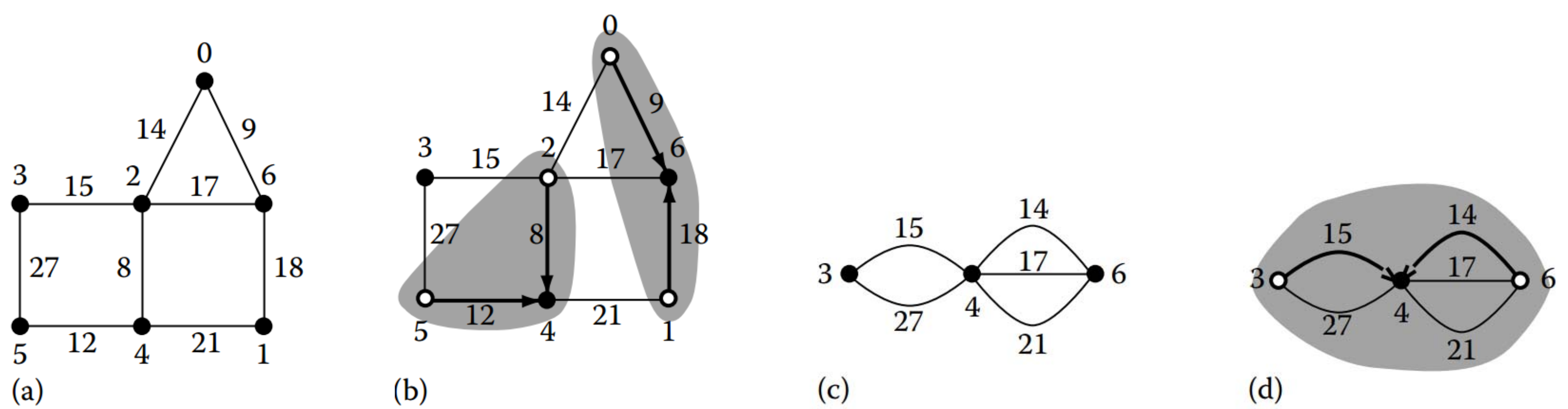
- Given an undirected graph with edge weights, an MST is a connected subgraph containing all the vertices, which has minimum total weight.
- Can be solved in O(m + n log n) time sequentially by a greedy algorithm.
- Key property is that for any set of vertices W, the minimum cost edge from W to V \ W is in the MST.
  - So for any vertex v, min cost edge containing v is in MST.
- Will describe a parallel MST algorithm based on the randomized parallel algorithm for connected component.



Source: Introduction to Algorithms, Cormen et al.

Randomized parallel algorithm

- Each node randomly chooses to be a parent or child node.
- Each child node u finds min weight incident edge (u,v), and points to v if v is parent.
  - This forms a set of stars with parents as centers.
  - If v isn't a parent, u forms its own star.
- Contract each star to the parent, and run algorithm recursively.
- Finding min weight incident edge quickly is a little tricky.
  - One possibility is to use priority CRCW.
  - Presort the edges by nondecreasing weight. Then when each E proc (u,v) writes to u, min weight edge wins.



Complexity

- At least 1/4 of vertex processors become inactive each phase in expectation.
  - Given a node u and min weight edge (u,v), there's 1/4 probability u is child and v is parent.
- Thus, there are O(log n) phases with high probability.
  - Finding min weight incident edge takes O(1) time after presorting edge weights.
- Total time is O(log n), total work is O((m+n) log n).