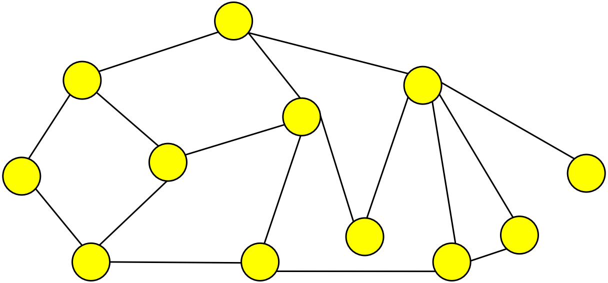
Massive Data Algorithmics

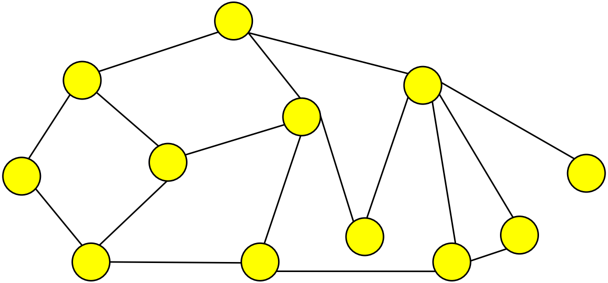
Lecture 9: Algorithms for trees

Graphs



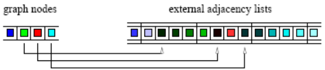
- Massive graphs
  - Web modeling: web crawling
  - Geographic information systems: Modeling terrains by graphs
- Representing graphs
  - Adjacency list
  - Unordered collection of edges

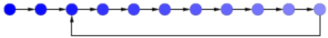
Graphs



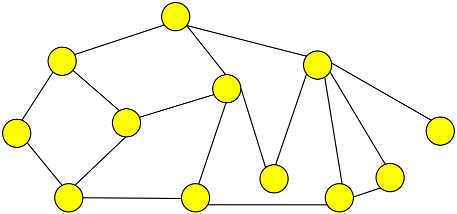
- Massive graphs
  - Web modeling: web crawling
  - Geographic information systems: Modeling terrains by graphs
- Representing graphs
  - Adjacency list
  - Unordered collection of edges

- Key difficulties in designing I/O-efficient graph algorithms
  - Nodes visited in unpredictable order. unstructured access to adjacency lists seems to need at least one I/O per node.
  - Remembering settled nodes requires extra data structures-algorithmic changes.

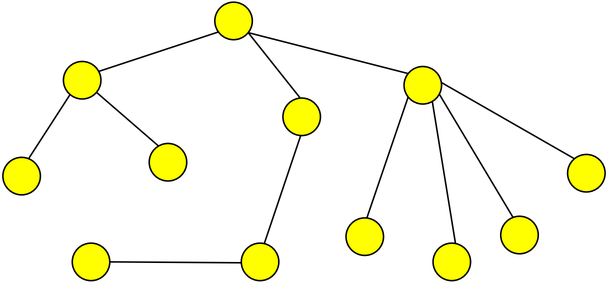




Graphs

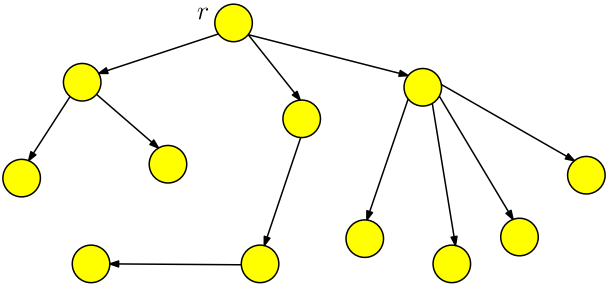


- Many results, many open questions.
- Undirected case often easier than directed cases.
- Dense graphs often easier than sparse graphs
- Special graph classes often easier
- General Methods: Time-forward processing, PRAM simulation, Graph reduction, ...
- Efficient solutions: MST, CC, Listranking, ...
- Still difficult: BFS, DFS, Shortest paths, ...



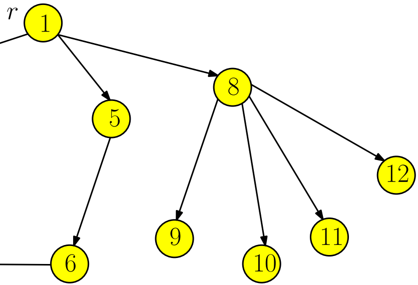
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



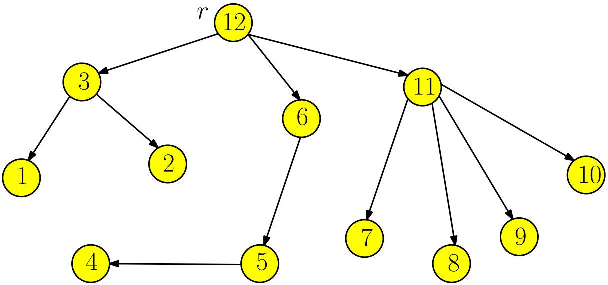
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



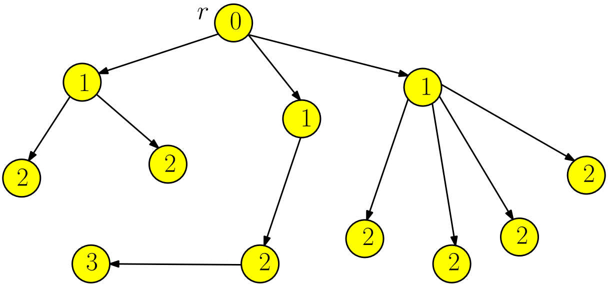
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



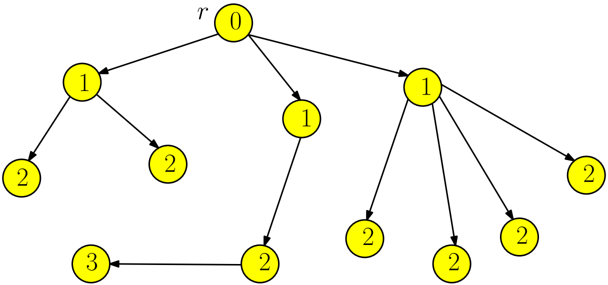
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



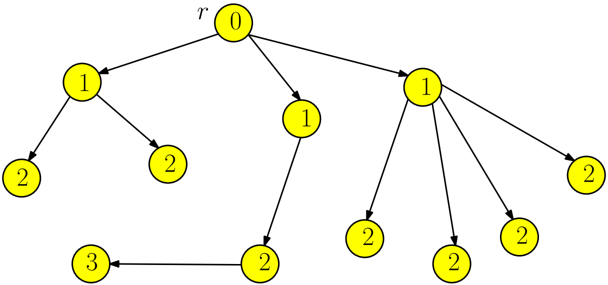
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



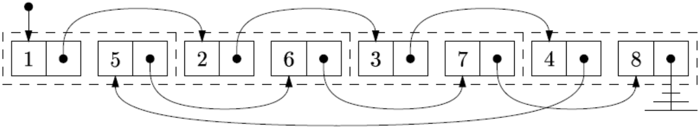
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Fundamental algorithms for trees



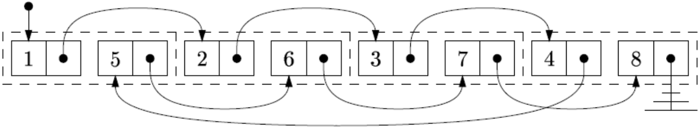
- Fundamental algorithms on tree $T = (V, E)$
  - Make rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Can be simply done with $O(|V|)$ I/Os
- Can be done in $O(\text{sort}(|V|))$?

Listranking



- Given a link list L, compute for every element of L its distance from the head of L.
- More General: each element v associated with $w(v)$. Compute $\rho(v)$ where $\rho(v) = \rho(\text{pred}(v)) \oplus w(v)$.

Listranking



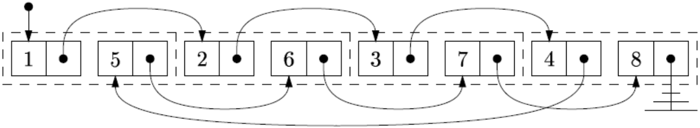
- Naive algorithms

Procedure NAÏVELISTRANKING

```
v ← h
ρ ← 0_l                      {0_l is the left-neutral element w.r.t. ⊕.}
while v ≠ nil do
    ρ ← ρ ⊕ ω(v)
    ρ(v) ← ρ
    v ← succ(v)
end while
```

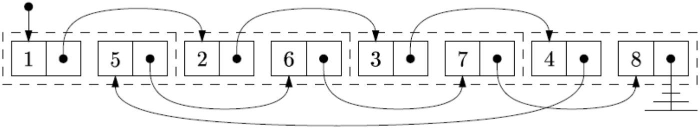
- $O(|V|)$ I/Os with LRU paging strategy

Listranking



- Maintained information for each node
  - Node id
  - Successor id
  - $w(v)$ (known) and $\rho(v)$ (to be computed)
  - extra data depending on applications

Listranking



- Overall strategy
  - If L fits into memory, load L to the memory.
  - Construct L' with size $2/3|L|$ with removing a large independent set I.
  - Updates the weight of elements in LI so that their weight ranks in L and L' are the same.
  - Recurse on L'
  - Compute the weight rank of elements in I by adding their weights to the weight ranks of their predecessors
- $\mathcal{I}(\mathcal{N}) = \mathcal{O}(\text{sort}(\mathcal{N}))$

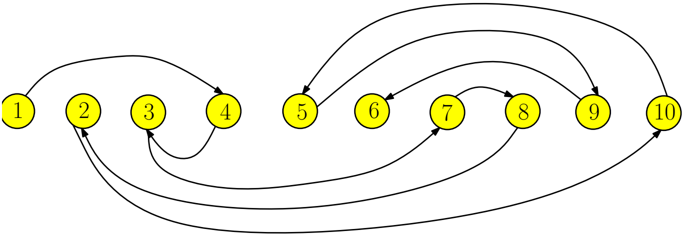
$$\mathcal{I}(N) = \begin{cases} \mathcal{O}(\text{scan}(N)) & \text{if } N \leq M \\ \mathcal{I}(\frac{2}{3}N) + \mathcal{O}(\text{sort}(N)) & \text{if } N > M \end{cases}$$

Listranking

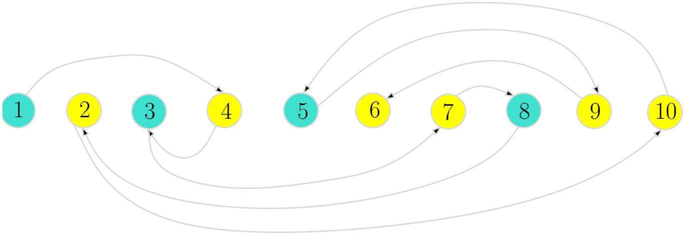
Procedure FastListRanking

if $|L| \le M$ **then**
  Load list L into main memory, and use procedure NaïveListRanking to compute the ranks of all elements in L.
else
  Find an independent set I of size at least $N/3$ in L.
  for all $v \in L \setminus I$ **do**
    $\mathrm{succ}_{L'}(v) \leftarrow \mathrm{succ}_L(v)$
    $\rho_{L'}(v) \leftarrow \rho_L(v)$
  end for
  for all $v \in I$ **do**
    if $\mathrm{succ}_L(v) \neq$ **nil then**
      $\omega_{L'}(\mathrm{succ}_L(v)) \leftarrow \omega_L(v) \oplus \omega_L(\mathrm{succ}_L(v))$
    end if
  end for
  for all $v \notin I$ **do**
    if $\mathrm{succ}_L(v) \neq$ **nil and** $\mathrm{succ}_L(v) \in I$ **then**
      $\mathrm{succ}_{L'}(v) \leftarrow \mathrm{succ}_L(\mathrm{succ}_L(v))$
    end if
  end for
  Let L' be the list defined by the vertices in $L \setminus I$, pointers $\mathrm{succ}_{L'}(v)$ and weights $\omega_{L'}(v)$.
  Recursively apply procedure FastListRanking to list L'. Let $\rho_{L'}(v)$ be the rank assigned to every element v in $L \setminus I$.
  for all $v \notin I$ **do**
    $\rho_L(v) \leftarrow \rho_{L'}(v)$
    if $\mathrm{succ}_L(v) \neq$ **nil and** $\mathrm{succ}_L(v) \in I$ **then**
      $\rho_L(\mathrm{succ}_L(v)) \leftarrow \rho_L(v) \oplus \omega_L(\mathrm{succ}_L(v))$
    end if
  end for

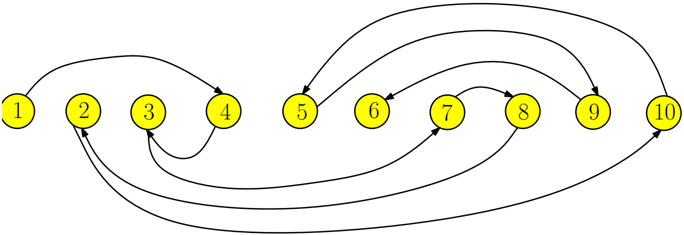
Listranking



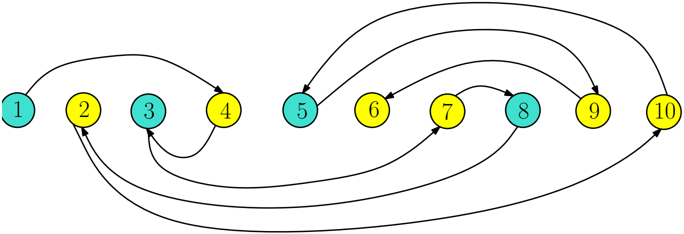
- Line 4: $O(\text{sort}(N))$



Listranking

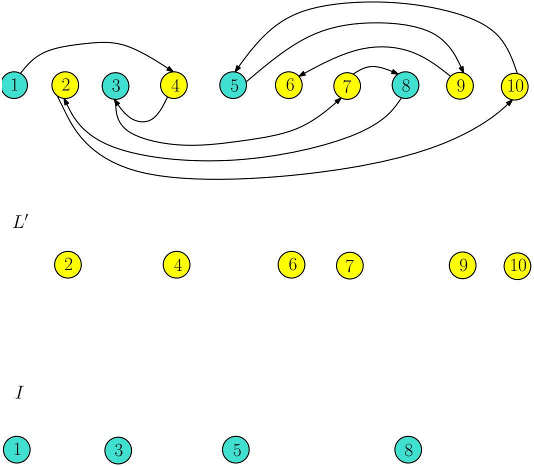


- Line 4: $O(\text{sort}(N))$

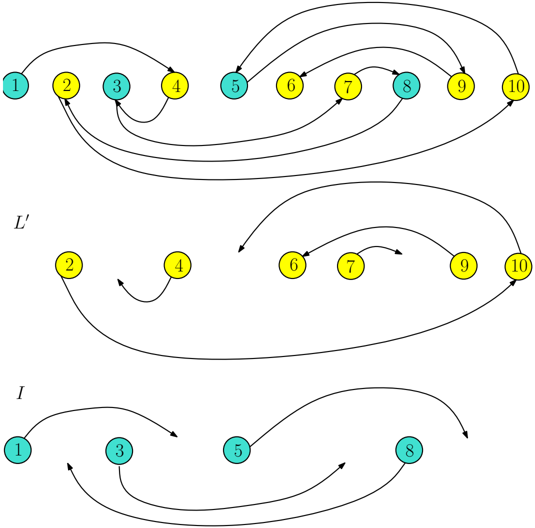


Listranking

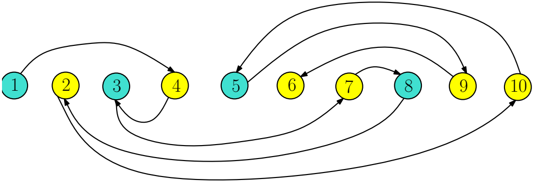
- Line 5-8: $O(\text{scan}(N))$



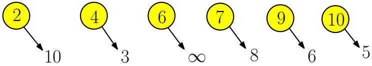
- Line 5-8: $O(\text{scan}(N))$



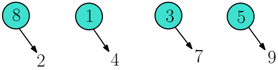
- Line 9-13: $O(\text{sort}(N))$



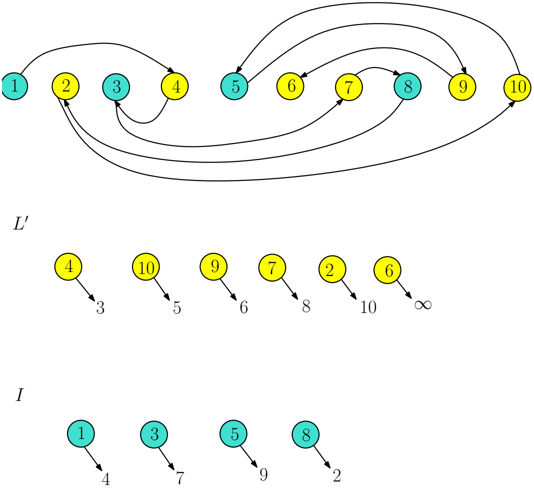
L'



I



- Line 14-18: $O(\text{sort}(N))$



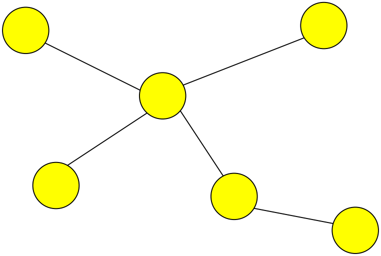
Listranking

- Line 19-20: $O(I(2/3N))$

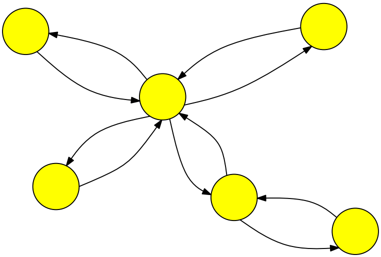
- Line 21-26: $O(\text{sort}(N))$
  - Sort L' based on their weight ranks
  - Sort I based on the weight ranks of their successors

Euler Tour

- Replace $\{v, w\}$ with directed edges (v, w) and (w, v)
- $\forall v \in T$:
  - Let incoming edges be $e_1, \cdots, e_k$ and outgoing edges be $e'_1, \cdots, e'_k$ where e_i and e'_i have the same endpoints
  - edge e_i is succeeded by edge $e'_{i \bmod k}$

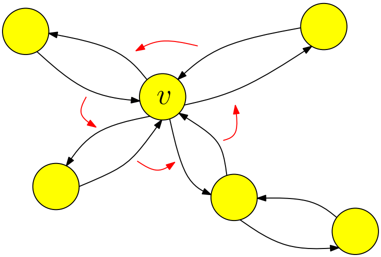


- Replace $\{v, w\}$ with directed edges (v, w) and (w, v)
- $\forall v \in T$:
  - Let incoming edges be $e_1, \cdots, e_k$ and outgoing edges be $e'_1, \cdots, e'_k$ where e_i and e'_i have the same endpoints
  - edge e_i is succeeded by edge $e'_{i \bmod k}$



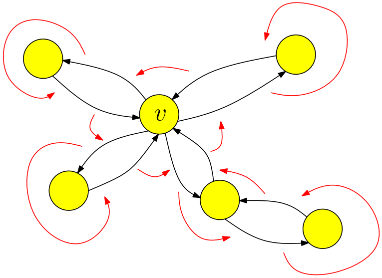
Euler Tour

- Replace $\{v, w\}$ with directed edges (v, w) and (w, v)
- $\forall v \in T$:
  - Let incoming edges be $e_1, \cdots, e_k$ and outgoing edges be $e'_1, \cdots, e'_k$ where e_i and e'_i have the same endpoints
  - edge e_i is succeeded by edge $e'_{i \bmod k}$



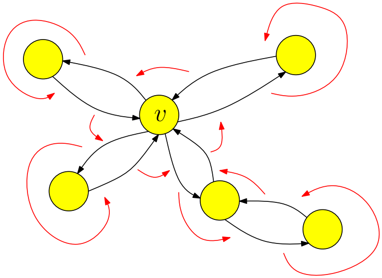
Euler Tour

- Replace $\{v, w\}$ with directed edges (v, w) and (w, v)
- $\forall v \in T$:
  - Let incoming edges be $e_1, \cdots, e_k$ and outgoing edges be $e'_1, \cdots, e'_k$ where e_i and e'_i have the same endpoints
  - edge e_i is succeeded by edge $e'_{i \bmod k}$



Euler Tour

- Adjacency list representation
  - Euler tour: $O(\text{scan}(N))$
- Unorderd collection of edges
  - Euler tour: $O(\text{sort}(N))$



Rooting a tree

- A tree can be rooted in $O(\text{sort}(N))$ I/Os

1: Compute an Euler tour L of tree T
2: Compute the rank of every edges e in L
3: for every edges $\{u,w\} \in T$ do
4: Store the ranks of edges (v,w) and (w,v) in L with $\{u,w\}$

Labeling rooted trees

- Labeling
  - Preorder
  - Postorder
  - Depth
- Procedure LabelTree
  1: Compute an Euler tour L of tree T that start at the root of T
  2: Assign appropriate weights to the edges in the Euler tour
  3: Compute the weighted rank of each edges in L
  4: Extract a labeling of the vertices of T from these ranks

Weight assigning

- Depth

$$w(e) = \begin{cases} 1 & \text{if } v = p(w) \\ -1 & \text{if } w = p(v) \end{cases}$$

- Preorder

$$w(e) = \begin{cases} 1 & \text{if } v = p(w) \\ 0 & \text{if } w = p(v) \end{cases}$$

Evaluating Directed Acyclic Graphs

- Given a DAG $G = (V, E)$
  - Each vertex is associated with $w(v)$ (known) and $(\rho(v))$ (to be computed)
  - $\rho(v)$ depends on the in-neighbors $u_1, \cdots, u_k$ of v
- Listranking is a special case
- Two assumptions to get efficient solution
  1. Vertices are given in a topological sort, otherwise $\Omega(|V|)$ I/Os are needed to topologically sort vertices
  2. If the in-degree is unbounded, computation of $\rho(v)$ from its in-neighbors $u_1, \cdots, u_k$ can be done in $O(\text{sort}(k))$ I/Os

  * Since Listranking is so restricted without two above assumptions we get efficient solution

Time-Forward Processing

- Procedure TimeForwardProcessing

```
$Q \leftarrow \emptyset$
For every vertex $v \in G$ in topologically sorted order do
    Let $u_1, \cdots, u_k$ be in-neighbors of $v$
    Retrieve $\rho(u_1), \cdots, \rho(u_k)$ from $Q$ using $k$ DeleteMin operations
    Compute $\rho(v)$ from $w(v)$ and $\rho(u_1), \cdots, \rho(u_k)$
    Let $w_1, \cdots, w_\ell$ be out-neighbors of $v$
    Insert $\ell$ copies of $\rho(v)$ into priority queue $Q$. Give the i-th copy
    priority $w_i$
```

- A DAG G can be evaluated in $O(\text{sort}(E))$ I/Os if vertices are given a topologically sorted order

Maximal Independent Set

- Procedure MaximalIndependentSet

  1: $I \leftarrow \emptyset$
  2: Direct the edge of G from vertices with lower numbers to vertices with higher numbers
  3: Sort the vertices of G by their numbers and the edges by the number of their sources
  4: for every vertices $v \in G$ in sorted order
  4:     if no in-neighbor of v is in I then
  5:         add v to I

- Line 4-8 can be simulated using Time-Forward Processing
- A maximal independent set of a undirected graph G can be computed in $O(\text{sort}(|V| + |E|))$

Maximal Independent Set

- Any maximal independent set of a list L has size at leas $N/3$, since every vertex has at most two neighbors
- A maximal independent set of a list L can be computed in $O(\text{sort}(N))$

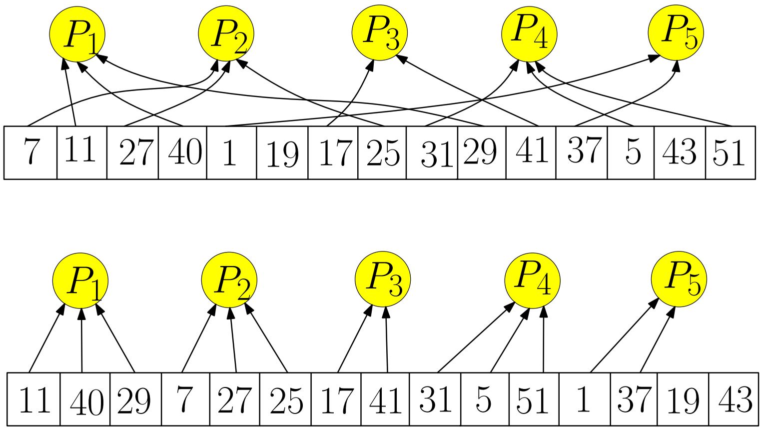
PRAM Simulation

- Parallel Random Access Machine (PRAM)
  - N processors
  - Shared Memory
- Read/write conflicts
  - Exclusive Read Exclusive Write (EREW)
  - Concurrent Read Exclusive Write (CREW)
  - Exclusive Read Concurrent Write (ERCW)
  - Concurrent Read Concurrent Write (CRCW)

PRAM Simulation

- Assumptions
  - N processors and N space
  - EREW strategy
  - In a single step, each PRAM processor reads $O(1)$ operands from memory, performs some computation, and then writes $O(1)$ results to memory.
- Simulation
  - Sort a copy of the contents of the PRAM memory based on the indices of the processors for which they will be operands in this step.
  - Scan this copy and perform the computation for each processor being simulated, and write the results to the disk as we do so
  - Sort the results of the computation based on the memory addresses to which the PRAM processors would store them and then scan the list and a reserved copy of memory to merge the stored values back into the memory.

PRAM Simulation



PRAM Simulation

- If a PRAM algorithm using $O(N)$ space and processors runs in T steps, the algorithm can be simulated using $O(T.\text{sort}(N))$ I/Os
- If every $O(1)$ steps, space and the number of processors decrease by a constant factor of N, the algorithm can be simulated in $O(\text{sort}(N))$ I/Os.

Summary: Algorithms for trees

- Listranking can be performed in $O(\text{sort}(N))$ I/Os
- The following algorithms can be done on trees using Listranking
  - Making rooted
  - Preorder ranking
  - Postorder ranking
  - Computing depth
- Techniques
  - Time-forward processing
  - PRAM simulation

References

- **I/O efficient graph algorithms**
  Lecture notes by Norbert Zeh.
  - Section 1-4