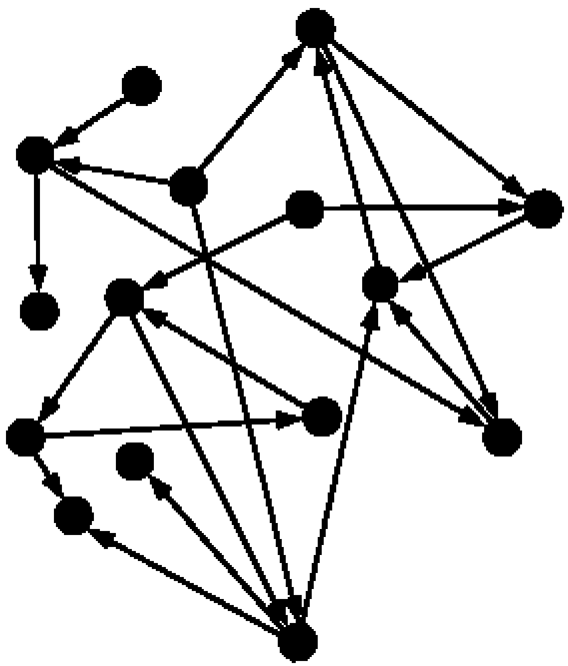
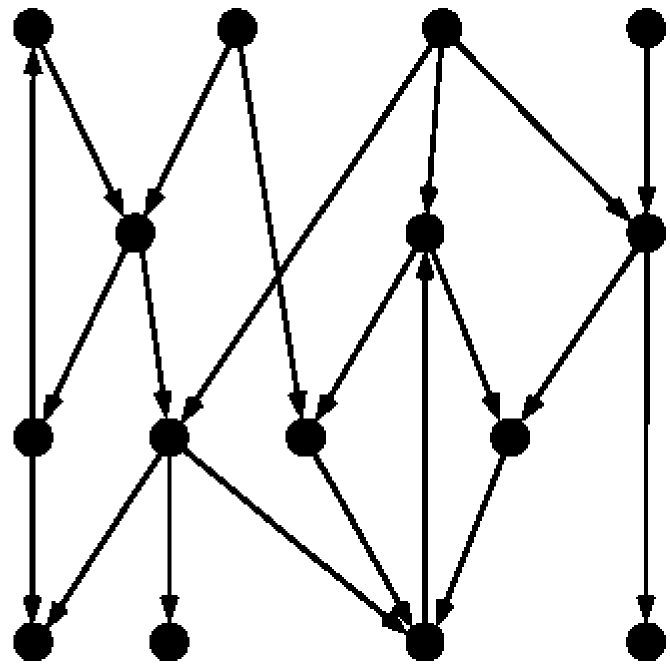
Layered Graph Drawing
(Sugiyama Method)

Drawing Conventions and Aesthetics



- a digraph



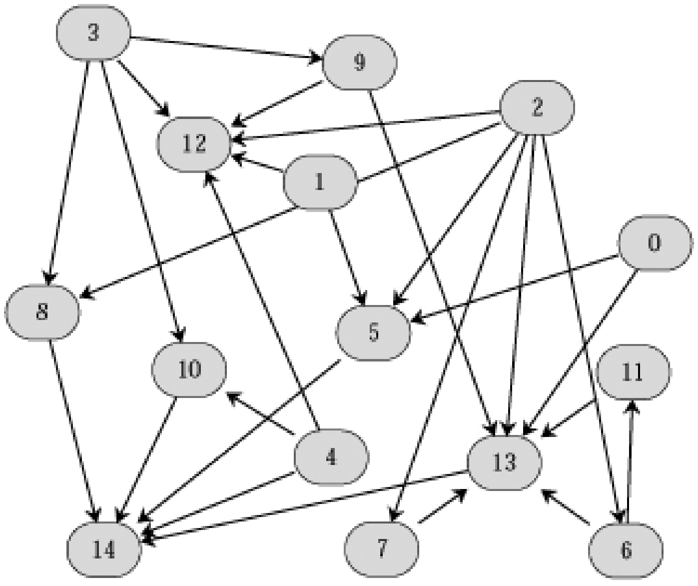
- A possible layered drawing

1. Edges pointing upward should be avoided.
2a. Nodes should be evenly distributed.
2b. Long edges should be avoided.
3. There should be as few edge crossings as possible.
4. Edges should be as straight/vertical as possible.

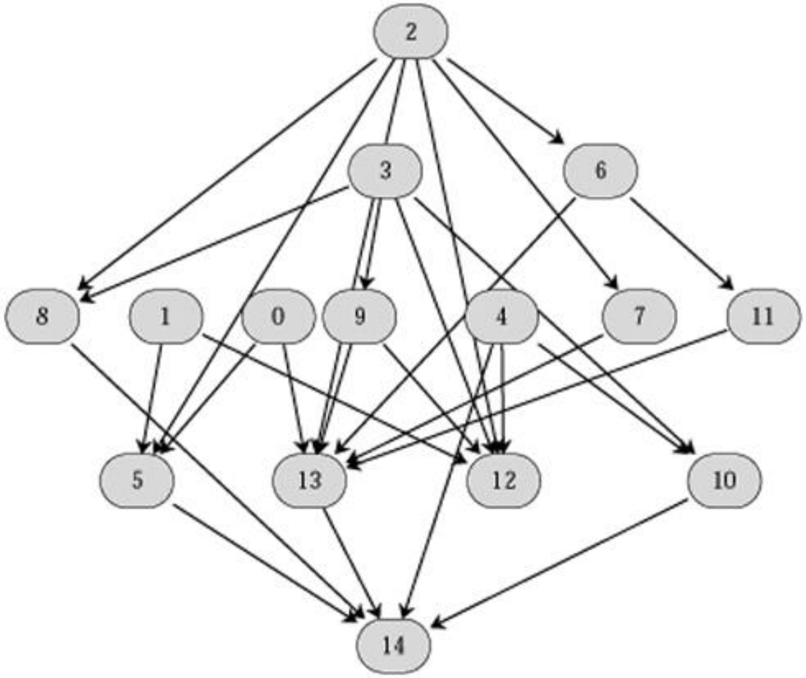
The Sugiyama method

- **Layered networks are often used to represent dependency relations.**
- **Sugiyama *et al.* developed a simple method for drawing layered networks in 1979.**
- **Sugiyama's aims included:**
  - **few edge crossings**
  - **edges as straight as possible**
  - **nodes spread evenly over the page**
- **The Sugiyama method is useful for**
  - **dependency diagrams**
  - **flow diagrams**
  - **conceptual lattices**
  - **other directed graphs: acyclic or nearly acyclic.**

The Sugiyama method

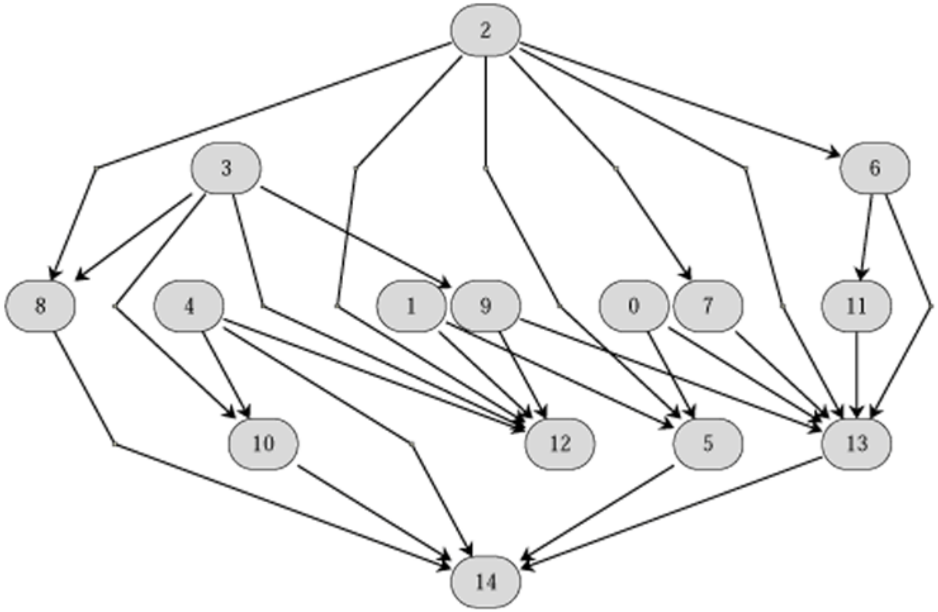


The Sugiyama method



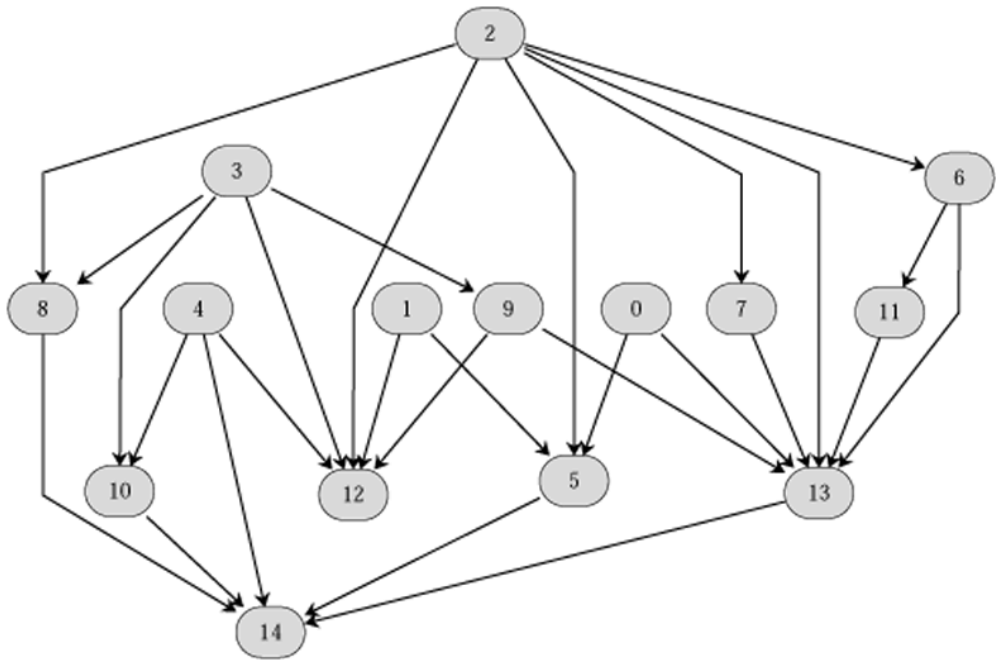
- Step 1
  Cycle Removal
- Step 2
  Layering

The Sugiyama method



- Step 1
  Cycle Removal
- Step 2
  Layering
- Step 3
  Node ordering

The Sugiyama method



- Step 1
  Cycle Removal
- Step 2
  Layering
- Step 3
  Node ordering
- Step 4
  Coordinate assignment

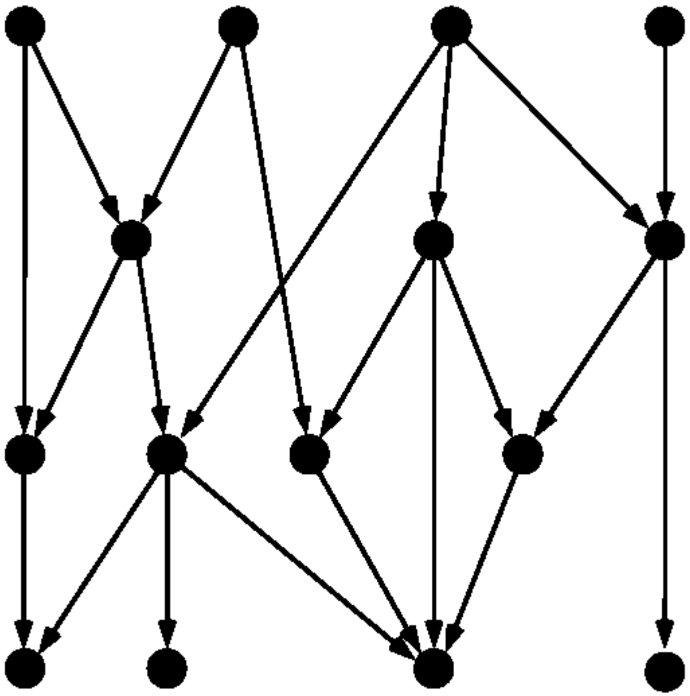
Layered Drawing of Digraphs

- Polyline drawings of digraphs with vertices arranged in horizontal layers
- Sugiyama, Tagawa and Toda '81
- Eades and Sugiyama '91
- Evolutionary algorithm approach of Branke et al.
- Magnetic field approach of Sugiyama and Misue.

- Attractive in practice: most graph drawing systemsinclude the Sugiyama method.

Step 1. Cycle Removal

- Input graph may contain cycles
  1. make an acyclic digraph by ***reversing*** some edges
  2. draw the acyclic graphs
  3. render the drawing with the original edge directions



- Acyclic graph by reversing two edges

Step 1. Cycle Removal

- Each cycle must have at least one edge against the flow
  - We need to keep the number of edges against the flow small
- Main problem: how to choose the set of edges R so that it is small

- Feedback arc set:
  - set of edges R whose ***reversing*** makes the digraph acyclic
- Feedback edge set:
  - set of edges whose ***removal*** makes the digraph acyclic

- Maximum acyclic subgraph problem
  - find a maximum set Ea such that the graph(V, Ea) contains no cycles : NP-hard
- Feedback arc set problem
  - find a minimum set Ef such that the graph(V, E \ Ef) contains no cycles : NP-hard

Step 1. Cycle Removal

- Edges in E \ Ea will be reversed

- Assume no two-cycles (or delete both two edges)

- Heuristics
  1. Fast heuristic
  2. Enhanced Greedy heuristic
  3. Randomized algorithm:[BS90]: O(mn) time

– Exact algorithm: [Grotschel et al 85, Reinelt 85]

1. Fast heuristic

- **Maximum acyclic subgraph problem**
  - **equivalent to unweighted linear ordering problem: find an ordering of the vertices, a mapping *o* such that the # of edges (u,v), *o*(u) > *o*(v) is minimized.**
- **Easiest heuristic**
  - **take an arbitrary ordering**
  - **then delete the edges (u,v) with *o*(u) > *o*(v)**
  - **May use given ordering: BFS or DFS**
  - **No performance guarantee: reverse |E|-|V|-1 edges (DFS)**
- **Heuristic that guarantees acyclic set of size at least ½|E| [BS90]**
  - **Delete for every vertex either incoming or outgoing edges**
  - **Linear time**

1. Fast heuristic [BS90]

Algorithm 8: A Greedy Algorithm

$E_a = \emptyset$;
foreach $v \in V$ **do**
  if $|\delta^+(v)| \geq |\delta^-(v)|$ **then**
    append $\delta^+(v)$ to E_a;
  else
    append $\delta^-(v)$ to E_a;
  delete $\delta(v)$ from G;

2. Enhanced greedy heuristic

- Feedback set problem: equivalent to finding a vertex sequence with as few leftward edges as possible
  - S=(v1, v2, …, vn): vertex sequence of a digraph G
  - Leftward edge: (vi, vj) with i > j
  - set of leftward edges for a vertex sequence forms a feedback set

Greedy Cycle Removal

- Greedy cycle removal heuristic [Eades et al 93]
  - Source & sink play a special role: edges incident to source & sink cannot be part of a cycle
  - Successively remove vertices from G
  - Add each in turn, to one of two lists Sl & Sr, either the end of Sl or the beginning of Sr
  - Greedy: choice of vertices to be removed and the choice of the list to be added

Greedy Cycle Removal

- **Greedy Cycle Removal [Eades et al 93]**
  - **All sinks (sources) should be added to Sr (Sl)**
  - **Choose a vertex u whose *outdeg(u)-indeg(u)* is maximized and add to Sl**
  - **performance**

$$|E_a| \geq \frac{|E|}{2} + \frac{|V|}{6}.$$

  - **Can be implemented in linear time and space**
  - **Sparse graph: Ea with at least 2/3|E|**

Greedy Cycle Removal

Algorithm 9: An Enhanced Greedy Heuristic

$E_a = \emptyset$;
while *G is not empty* **do**
1   **while** *G contains a sink v* **do**
        add $\delta^-(v)$ to E_a and delete v and $\delta^-(v)$ from G;
2   delete all isolated vertices from G;
3   **while** *G contains a source v* **do**
        add $\delta^+(v)$ to E_a and delete v and $\delta^+(v)$ from G;
4   **if** *G is not empty* **then**
        let v be a vertex in G with maximum value $|\delta^+(v)| - |\delta^-(v)|$;
        add $\delta^+(v)$ to E_a and delete v and $\delta(v)$ from G;

Analysis

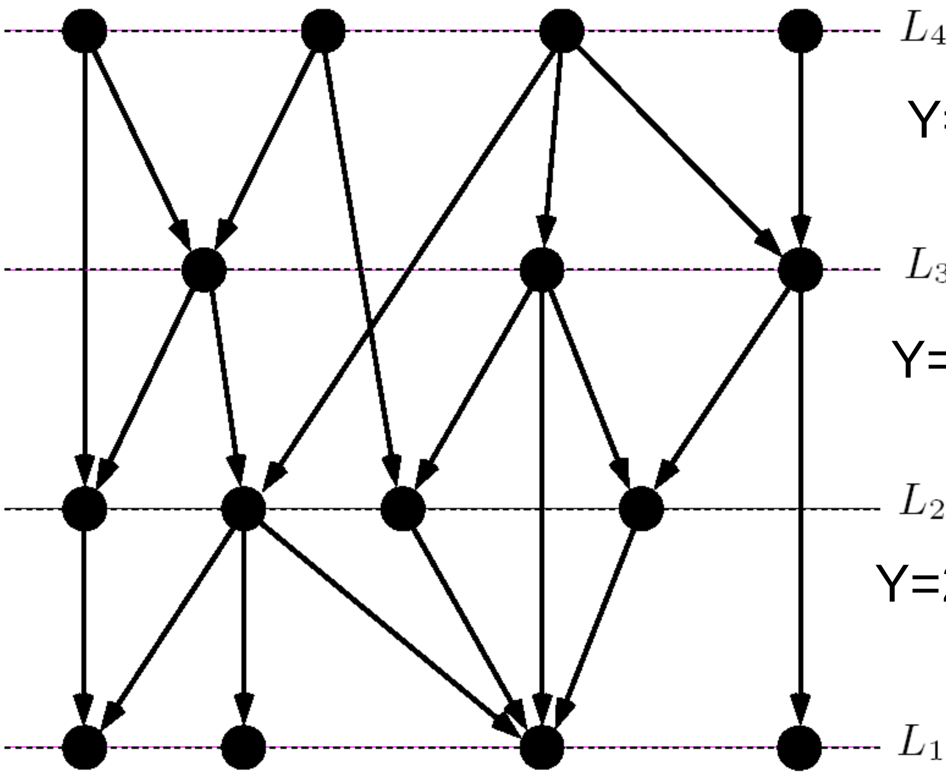
- Delete all two cycles before applying Greedy-cycle-removal
  - Two-cycle: a directed cycle with two edges
- [Theorem] G: connected digraph with n vertices & m edges without two cycles. Greedy-Cycle-Removal computes a vertex sequence S of G with at most m/2 – n/6 leftward edges
- [Theorem] Greedy-Cycle-Removal can be implemented in linear time & space
- Simple & speedy
- Sparse graph [EL95]

  [Theorem] G: connected digraph with n vertices & m edges without two cycles. Each vertex of G has total degree at most 3. Greedy-Cycle-Removal computes a vertex sequence S of G with at most m/3 leftward edges

Step 2. Layer Assignment

- Layering: partition V into L1, L2, …, Lh
- Layered (di)graph: digraph with layers
- Height h: # of layers
- H-layered graph: digraph with height h
- Width w: # of vertices with largest layer
- Span of an edge
- Proper digraph: no edge has a span > 1

- Some application, vertices are preassigned to layers
- However, in most applications, we need to transform an cyclic digraph into a layered digraph



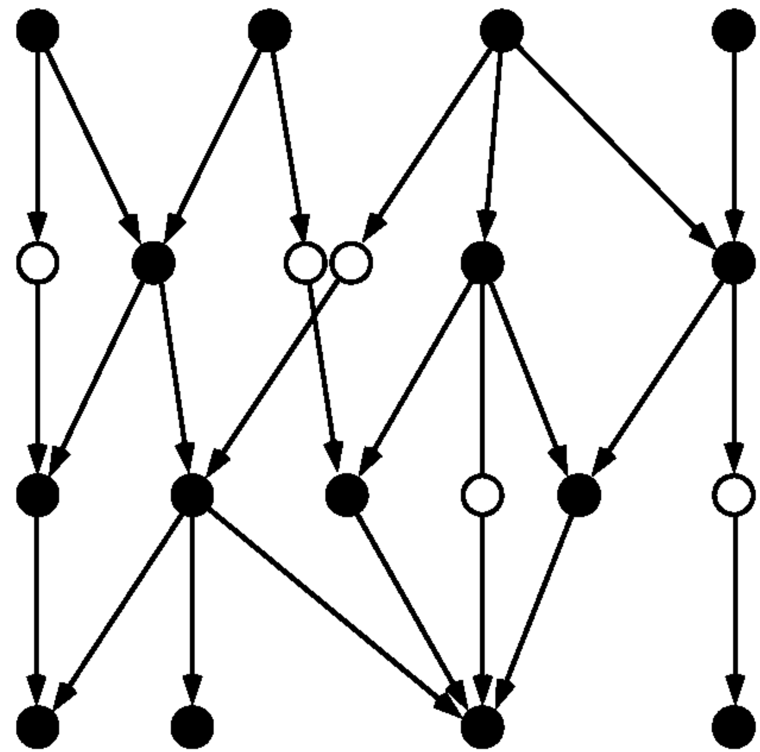
Y=4

Y=3

Y=2

Y=1

- Layering



- Introducing dummy vertices

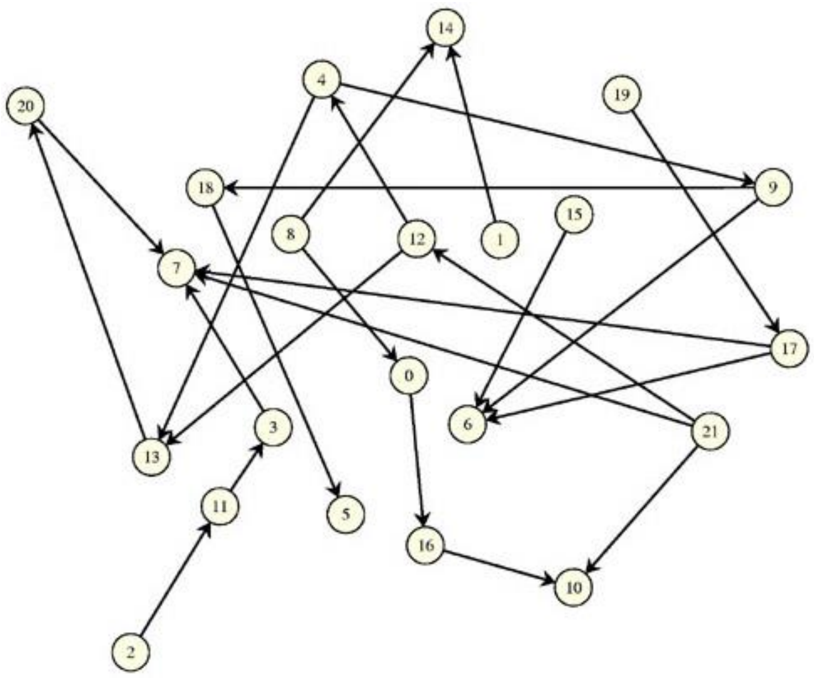
Step 2. Layer Assignment

- **Requirements**
  1. Layered digraph should be compact: height & width
  2. The layering should be proper: add dummy vertices
  3. The number of dummy vertices should be small
     - A. time depends on the total number of vertices
     - B. bends in the final drawing occur only at dummy vertices
     - C. the number of dummy vertices on an edge measures the y extent of the edge: avoid long edge.
- **Methods**
  1. Longest path layering: minimize height
  2. Layering to minimize width
  3. Minimize the number of dummy vertices

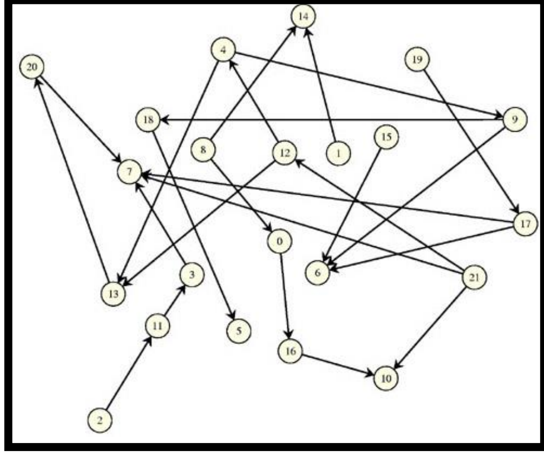
Three Layering Algorithms

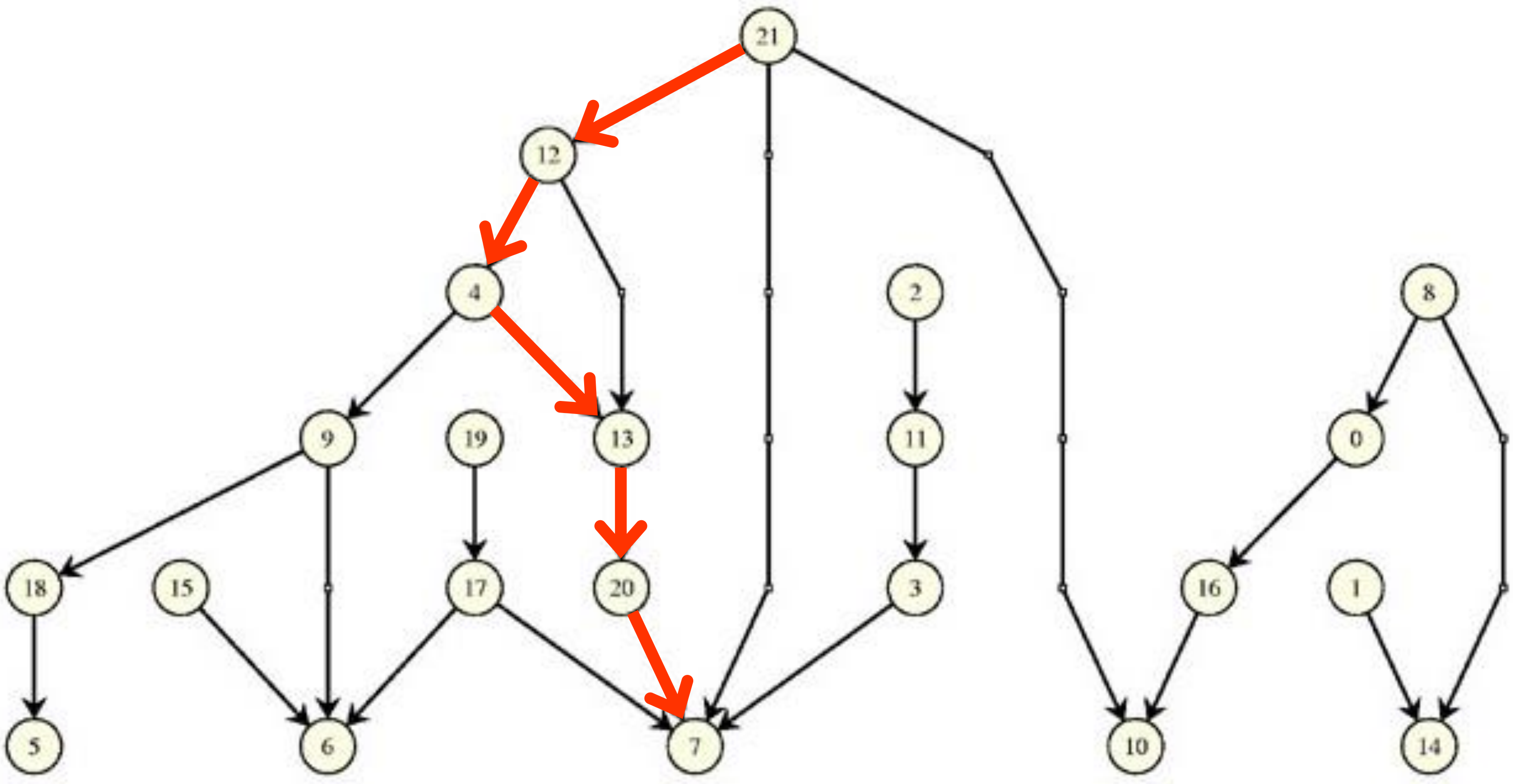
- **Longest Path**
- **Coffman-Graham**
- **Network Simplex**

Grafo1012 (Di Battista *et al., Computational Geometry: Theory and Applications,* (7), 1997)

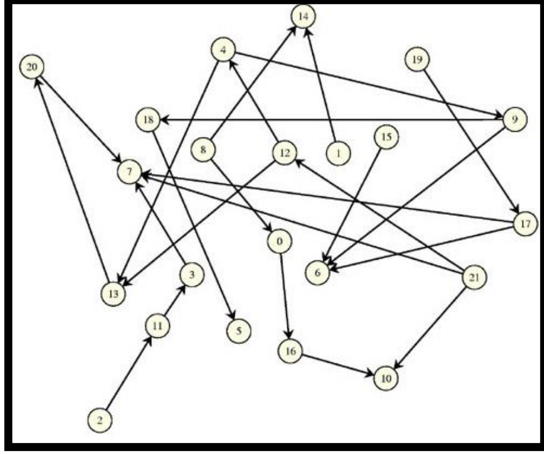


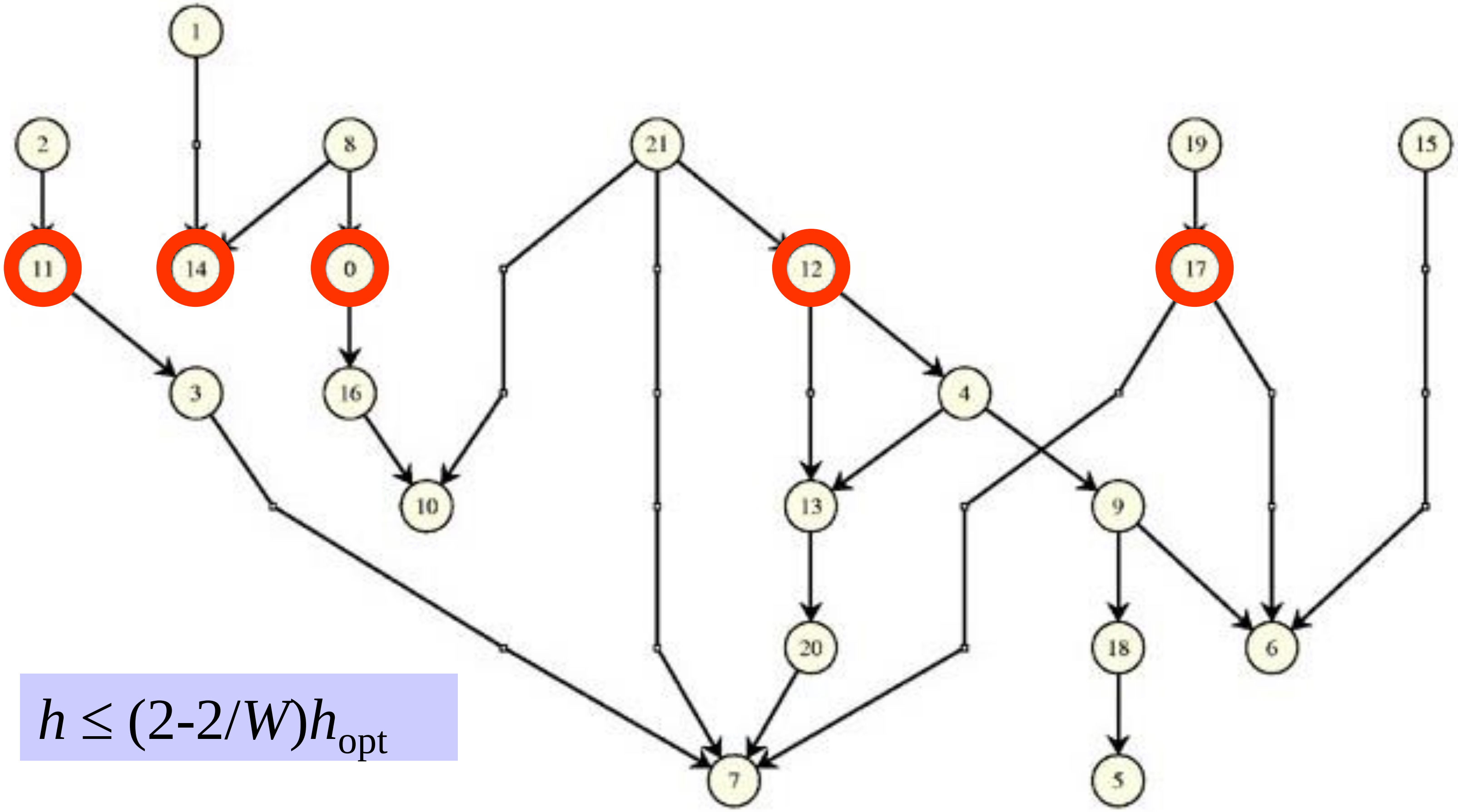
Longest Path Layering





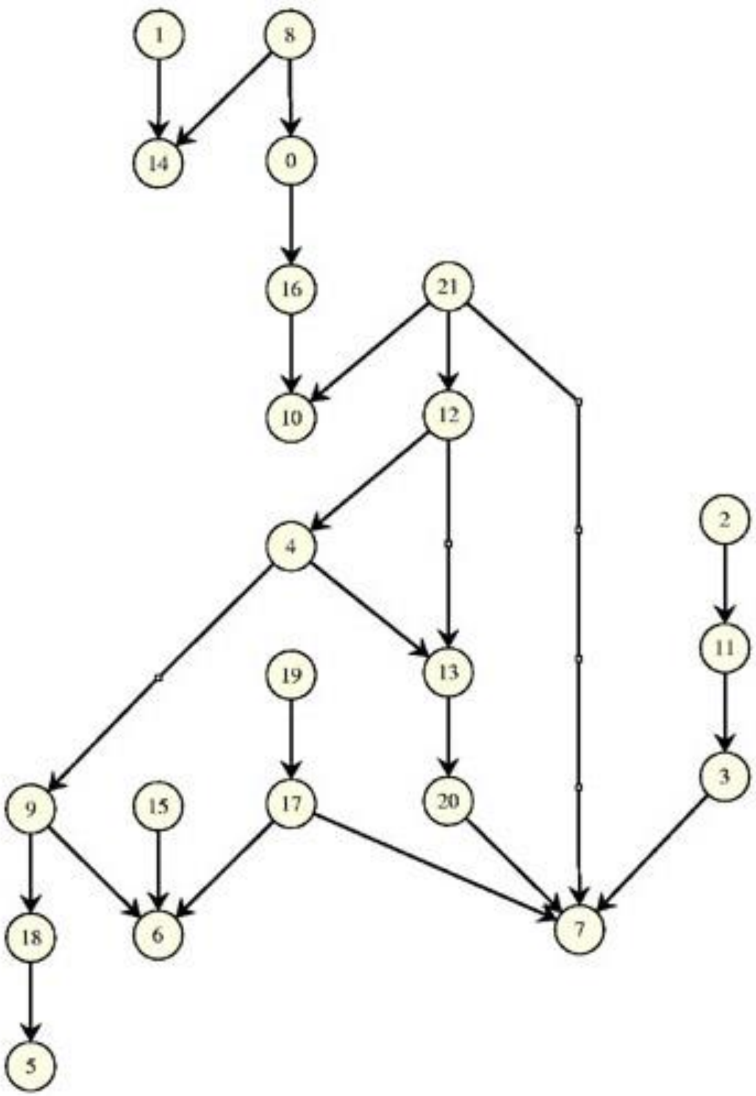
Coffman-Graham Layering
(1972)

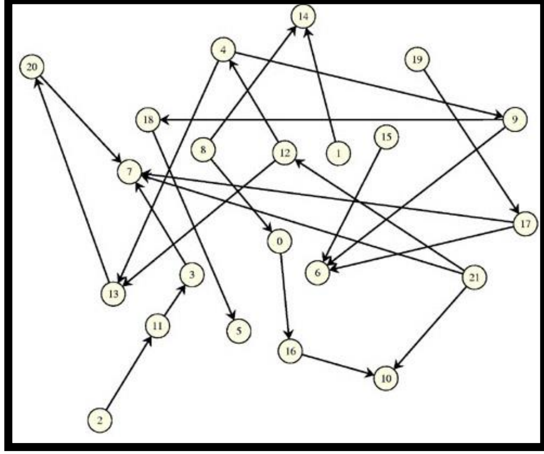




$$h \leq (2\text{-}2/W)h_{\mathrm{opt}}$$

Network Simplex Layering (AT&T, 1993)





ILP formulation

$$\min \sum_{(u,v)\in E} (y(u) - y(v))$$

$$y(u) \in Z, \forall u \in V,$$

$$y(u) \geq 1, \forall u \in V,$$

$$y(u) - y(v) \geq 1, \forall (u,v) \in E.$$

1. Longest path layering

- Minimizing the height

- Place all sinks in layer L_1
- Each remaining vertex v is placed in layer L_{p+1}, where the longest path from v to a sink has length p

$$y(u) := \max\{i \mid v \in N^+(u) \text{ and } y(v) = i\} + 1$$

$$N^+(u) := \{v \in V \mid \exists (u, v) \in E\}$$

- Can be computed in linear time
- Main drawback: too wide

2. Layering to minimize width

- Finding a layering with minimum height subject to a maximum width constraint: *Precedence-constrained multiprocessor scheduling problem* -> NP-complete [GJ79]

- Coffman-Graham Layering
  - Input: reduced graph G (no transitive edges) and W
  - Output: layering of G with width at most W
  - Aim: ensure the height of the layering is kept small [LS77]
  - Two phases
    1. Order the vertices
    2. Assign layers
- Width: does not count dummy vertices

Coffman-Graham Layering

- **Simple lexicographic order:**

$S \prec T$ if either

1. $S = \emptyset$ and $T \neq \emptyset$, or
2. $S \neq \emptyset$, $T \neq \emptyset$ and $\max(S) < \max(T)$, or
3. $S \neq \emptyset$, $T \neq \emptyset$, $\max(S) = \max(T)$ and $S \setminus \{\max(S)\} \prec T \setminus \{\max(T)\}$

- **First phase: lexicographical ordering**
- **Second phase: ensure that no layer receive more than W vertices**

- **[LS77] height is not too large** $h \leq (2 - \frac{2}{w})h_{opt}$

Coffman-Graham Layering

Algorithm 12: Coffman-Graham-Algorithm

foreach $v \in V$ **do** $\pi(v) := n + 1$;
for $i = 1$ ***to*** $|V|$ **do**
  choose a vertex v with $\pi(v) = n + 1$
    and minimum set $\{\pi(u) \mid (u, v) \in E\}$ with respect to $\prec$;
  $\pi(v) := i$;

$k := 1$; $L_1 := \emptyset$; $U := V$;
while $U \neq \emptyset$ **do**
  choose $u \in U$ such that every vertex in $\{v \mid (u, v) \in E\}$ is in $V \setminus U$
    and $\pi(u)$ is maximized;
  if $|L_k| < w$ *and* $N^+(u) \subseteq L_1 \cup L_2 \cup \ldots L_{k-1}$ **then**
    add u to L_k;
  else
    $k := k + 1$; $L_k := \{u\}$;
  delete u from U;

3. Minimizing # of dummy vertices

- one can compute a layering in polynomial time that minimizes the number of dummy vertices [GKNV93]
- f = $\Sigma_{(u,v)\in V}$ (y(u) - y(v) - 1)
- f: sum of vertical spans of the edges in the layering

- # of edges : (# of dummy vertices)

- Layer assignment problem is reduced to choosing y-coordinates to minimize f

- Integer linear programming problem

Remark

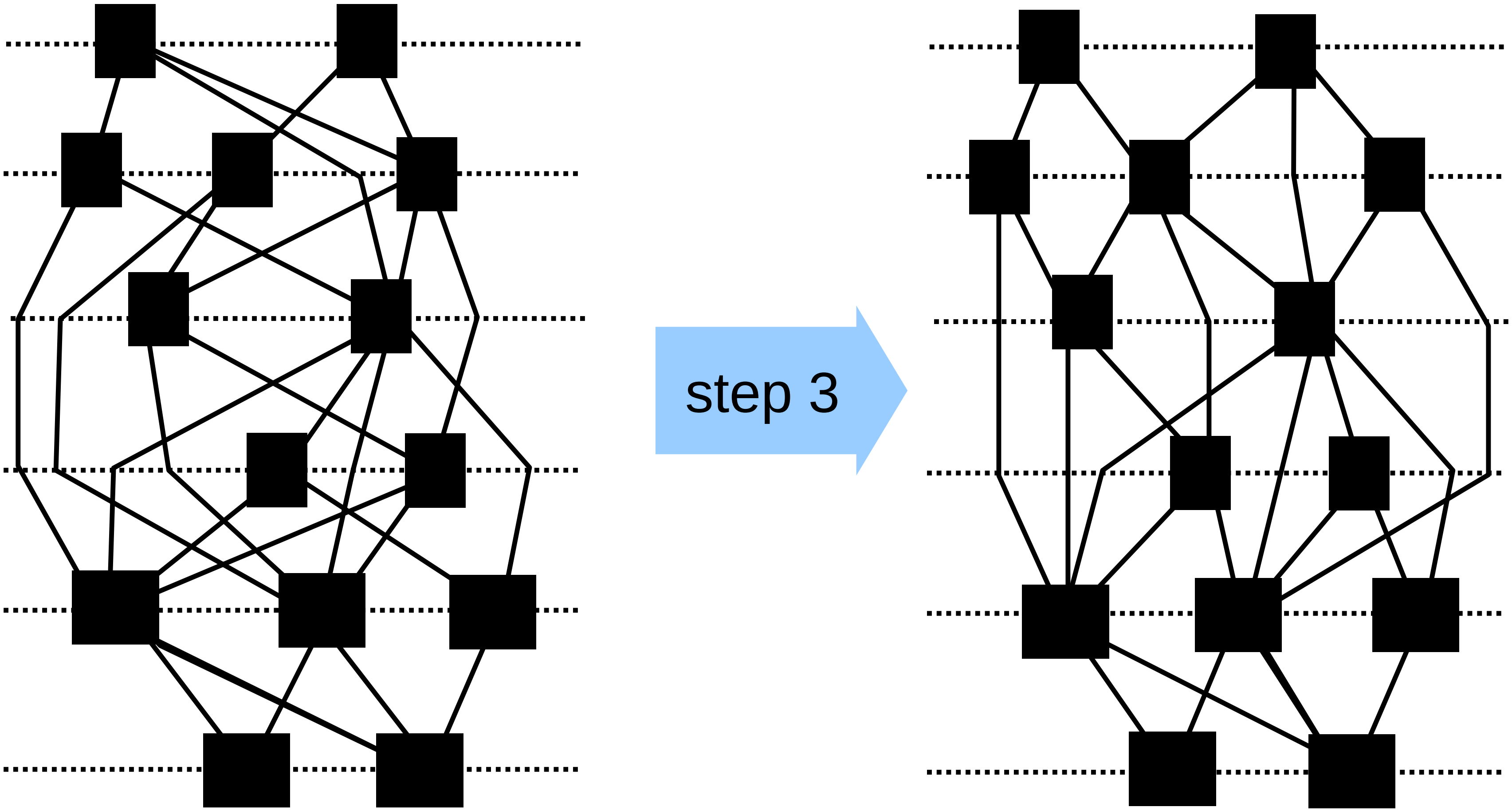
- **Methods**
  1. **Layering for general graphs [Sander 96]**
  2. **Minimizing the height: Longest path layering**
  3. **Layering with given width: Coffman-Graham algorithm: width is more important than height**
  4. **Minimizing the total edge span (# of dummy vertices) : relatively compact drawing**
- **[Sander 96]**
  1. **Calculate y by DFS or BFS**
  2. **Calculate minimum cost spanning trees**
  3. **Apply spring embedder**

Step 3. Crossing Reduction

- Input: proper layered graph
- # of edge crossings does not depend on the precise position of the vertices, but only the ordering of the vertices within each layer (combinatorial, rather than geometric)

- NP-complete, even for only two layers [GJ83]
- Many heuristics
  - Layer-by-layer sweep: two layer crossing problem
  - 1. Sorting
  - 2. Barycenter method
  - 3. Median method
  - 4. Integer programming method: exact algorithm

Crossing Reduction: ordering



21 edge crossings

5 edge crossings

Layer-by-layer sweep

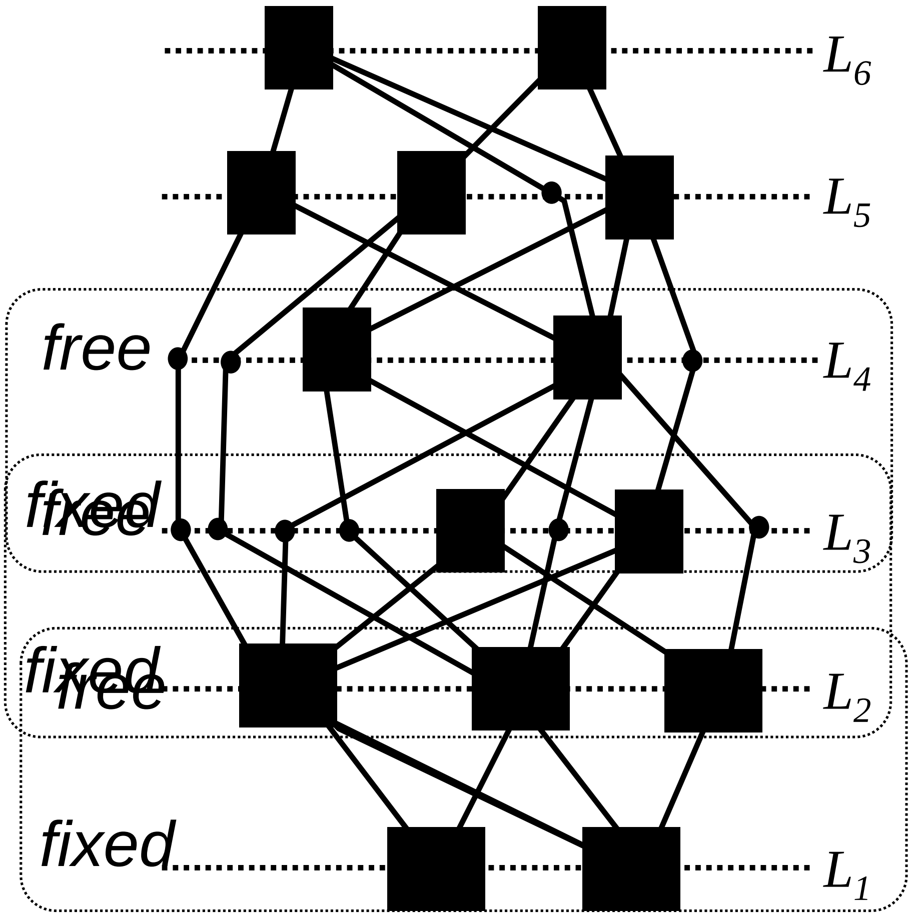
- A vertex ordering of layer L_1 is chosen
- For i = 2, 3, …, h
  - The vertex ordering of L_{i-1} is fixed
  - Reordering the vertices in layer Li to reduce edges crossings between L_{i-1} and L_i

- Two layer crossing problem:

  given a fixed ordering of L_{i-1}, choose a vertex ordering of Layer L_i to minimize # of crossings

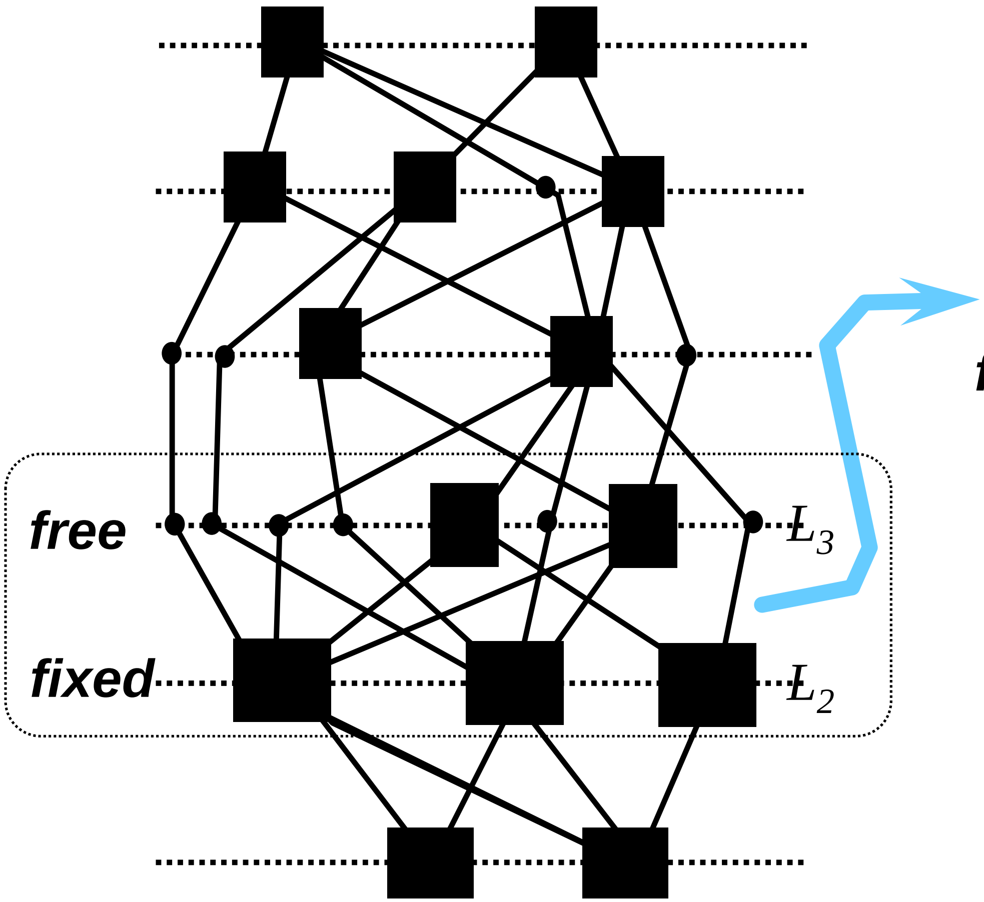
- Several variations: layer-by-layer sweep

Layer-by-layer sweep

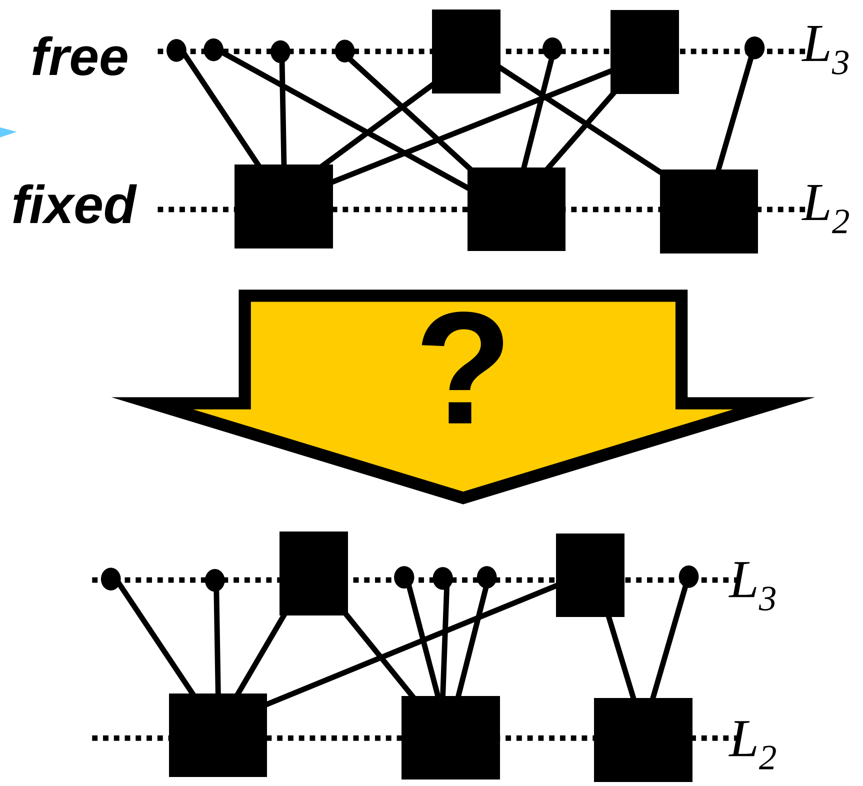


- Step 3 uses a "layer-by-layer sweep", from bottom to top.
- At each stage of the sweep, we:
  - hold one layer fixed, and
  - Re-arrange the nodes in the layer above to avoid edge crossings.

Two layer crossing problem

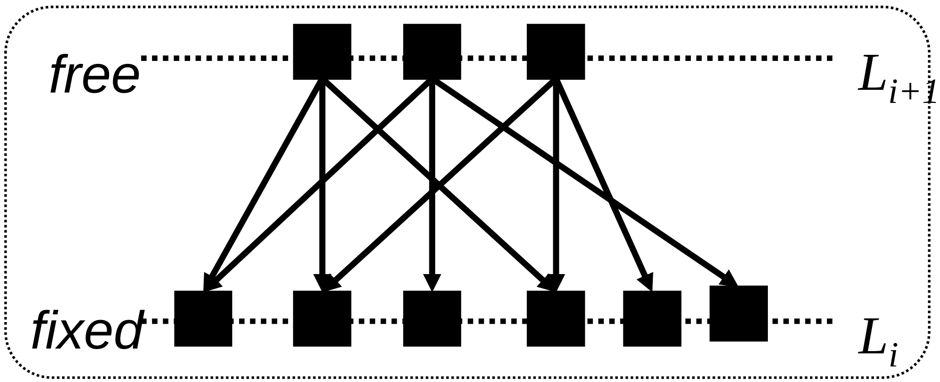


The difficult part is to re-arrange the free layer



Two layer crossing problem

- **The problem of finding an optimal solution is NP-hard.**



- ***Heuristics***
  1. ***Barycenter method:*** **place each free node at the barycenter of its neighbours.**
  2. ***Median method:*** **place each free node at the median of its neighbours.**

Two layer crossing problem

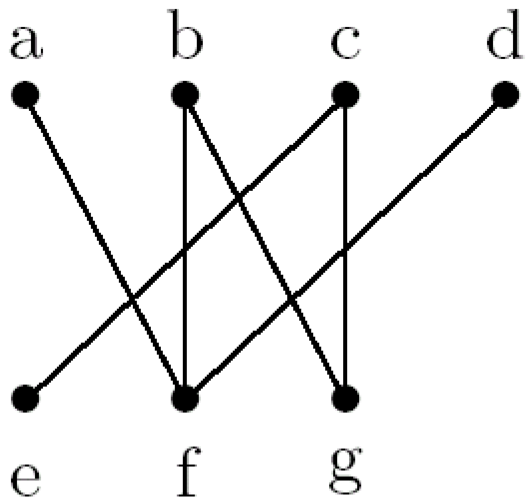
- given a two-layered digraph G=(L1,L2,E) and an ordering x1 of L1, find an ordering x2 of L2, such that cross(G,x1,x2) = opt(G,x1)
  - two-layered digraph G=(L1, L2, E): a bipartite digraph
  - cross(G, x1, x2): # of crossings in a drawing of G
  - opt(G,x1): min $_{x2}$ cross(G, x1, x2)
- NP-complete: [EW94]

- Simple observation: u and v are in L2

  the # of crossings between edges incident with u and edges incident with v depends only on the relative positions of u and v and not on the other vertices

Crossing number

- Crossing number c_{uv}
  - – # of crossings that edges incident to u make with edges incident v, when x2(u) < x2(v)
  - – # of pairs (u,w), (v,z) of edges with x1(z) < x1(w)



C	e	f	g
e	0	2	1
f	1	0	2
g	0	3	0

One-sided crossing minimization

$$\mathrm{opt}(G, \pi_1) = \min_{\pi_2} \mathrm{cross}(G, \pi_1, \pi_2)$$

Given a bipartite Graph $G = (V_1, V_2, E)$ and a permutation π_1 of V_1. Find a permutation π_2 of V_2 that minimizes the edge crossings in the drawing of G, i.e., $\mathrm{cross}(G, \pi_1, \pi_2) = \mathrm{opt}(G, \pi_1)$.

$$\mathrm{cross}(G, \pi_1, \pi_2) = \sum_{\pi_2(u) < \pi_2(v)} c_{uv} = \sum_{i=1}^{n_2 - 1} \sum_{j=i+1}^{n_2} c_{ij}$$

$$L = \sum_{\pi_2(u) < \pi_2(v)} \min\{c_{uv}, c_{vu}\}$$

1. Sorting Method

- **Aim: to sort the vertices in L2 into an order that minimizes # of crossings**
- **Naive algorithm: O(|E|)^2, can be reduced**
- ***Adjacent-Exchange***
  - **exchange adjacent pair of vertices using the crossing numbers, in a way similar to bubble sort**
  - **Scan the vertices of L2 from left to right, exchanging an adjacent pair u, v whenever $c_{uv} > c_{vu}$**
  - **O(|L2|^2) time**
- ***Split***
  - **quick sort: choose a pivot vertex p in L2, and place each vertex u to the left of p if $c_{up} < c_{pu}$, and to the right of p otherwise**
  - **Apply recursively to the left & right of p**
  - **O(|L2|^2) time in worst case; O((|L2|log (|L2|) in practice**

Adjacent-Exchange

Algorithm 13: greedy_switch

repeat

  for $u := 1$ *to* $|V_2| - 1$ **do**

    if $c_{u(u+1)} > c_{(u+1)u}$ **then**

      switch vertices at positions u and $u + 1$;

until *the number of crossings was not reduced*;

Split

Algorithm 14: split $(i, j : 1, \ldots, |V_2|)$

---

if $j > i$ **then**

$pivot := low := i;\ high := j;$

for $k := i + 1$ *to* j **do**

if $c_{k\,pivot} < c_{pivot\,k}$ **then**

$\pi(k) := low;\ low := low + 1;$

else

$\pi(k) := high;\ high := high - 1;$

$/*\ low == high\ */$

$\pi(pivot) := low;$

copy $\pi(i \ldots j)$ into $\pi_2(i \ldots j)$;

split $(i, low - 1)$;

split $(high + 1, j)$;

2. The Barycenter Method

- The most common method
- x-coordinate of each vertex u in L2 is chosen as the barycenter(average) of the x-coordinates of its neighbors
- x2(u) = bary(u) = 1/deg(u) Σ x1(v), v is a neighbor

$$\mathrm{bary}(u) = \frac{1}{\deg(u)} \sum_{v \in N(u)} \pi_1(v)$$

- If two vertices have the same barycenter, then order them arbitrarily
- Can be implemented in linear time

3. The Median Method

- Similar to the barycenter method
- x-coordinate of each vertex u in L2 is chosen as the median of the x-coordinates of its neighbors
- v1, v1, …, vj: neighbors of u with x1(v1) < x1(v2) < … < x1(vj)
  - med(u) = x1(vj/2)
  - if u has no neighbor, then med(u) = 0
- How to use med(u) to order the vertices in L2: sort L2 on med(u)
- If med(u) = med(v)
  - Place the odd degree vertex on the left of the even degree vertex
  - If they have the same parity, choose the order of u & v arbitrarily
- Can be computed using a linear-time median finding algorithm [AHU83]

Analysis

- **[Theorem]**

  if opt(G,x1)= 0, then bar(G,x1)=med(G,x1)=0

- **Performance guarantees**

  Theorem 1:

  The *barycenter method* is at worst O(sqrt(n)) times optimal.

  Theorem 2:

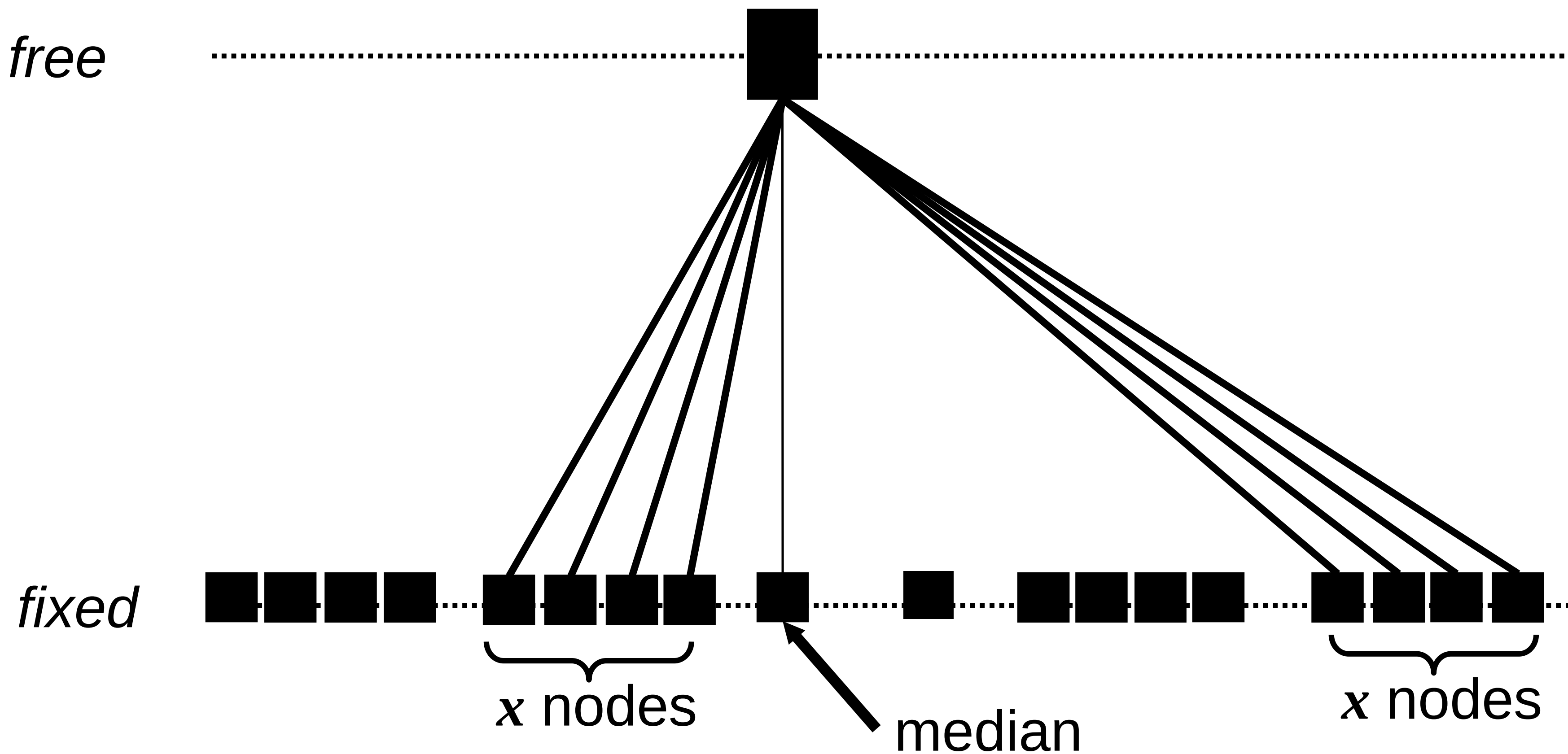
  The *median method* is at worst 3 times optimal.

$$(1)\ \frac{bar(G)}{opt(G)}\ is\ O\left(\sqrt{n}\right)$$

$$(2)\ \frac{med(G)}{opt(G)} \le 3$$

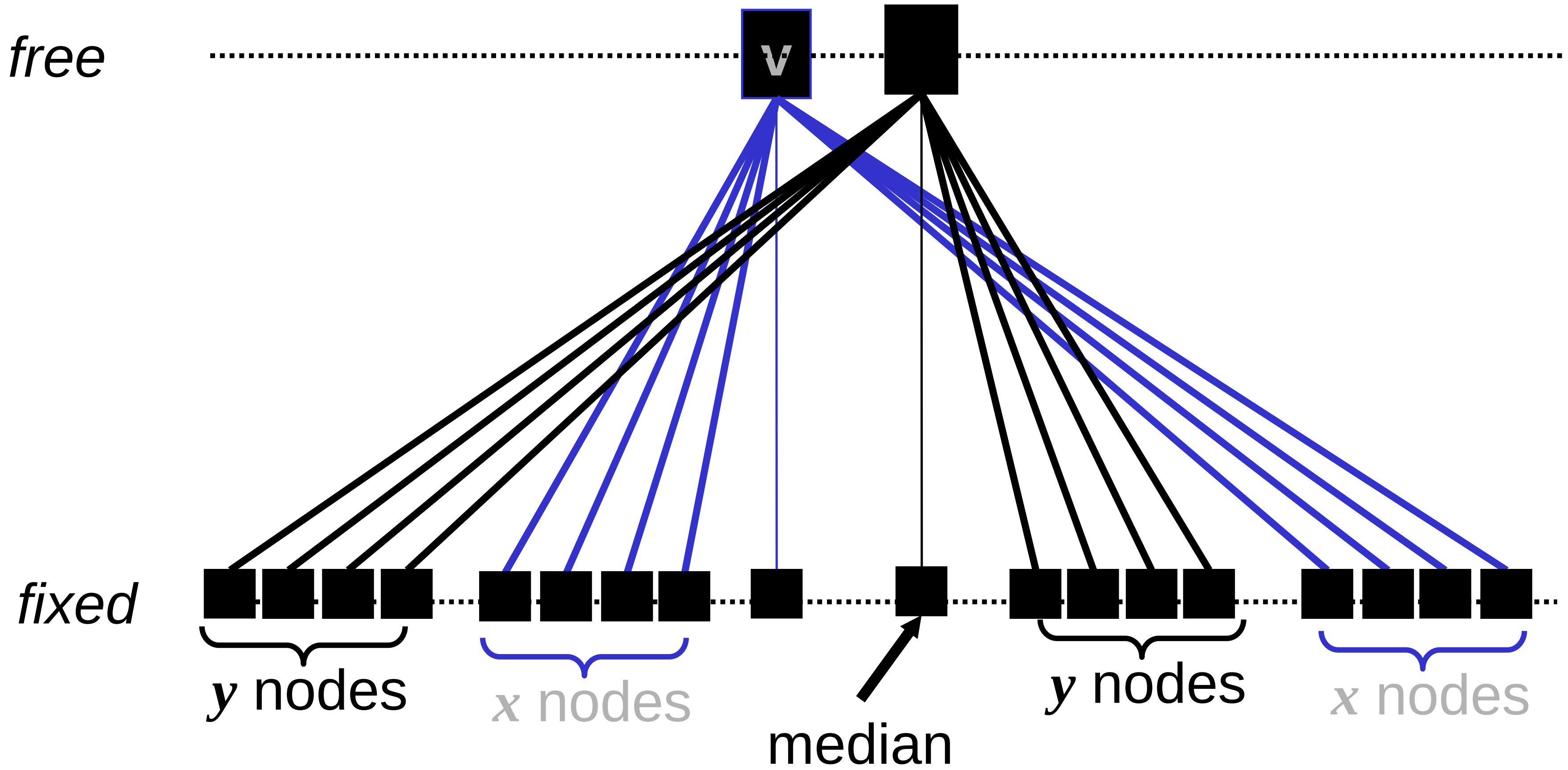
3. Median Method

- **Some intuition behind Theorem 2 (median method is at worst 3 times optimal).**



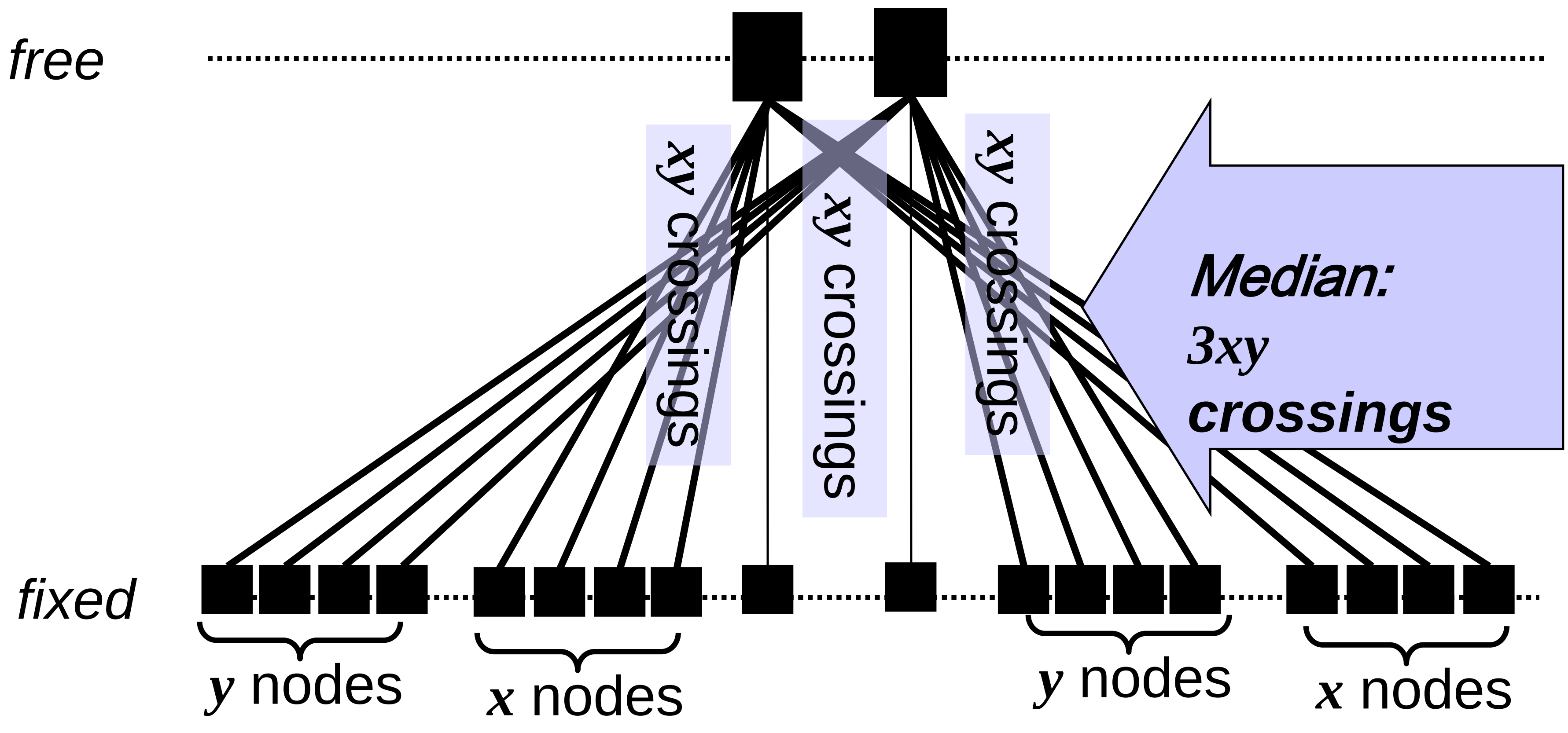
3. Median Method

■ **Some intuition behind Theorem 2 (median method is at worst 3 times optimal).**



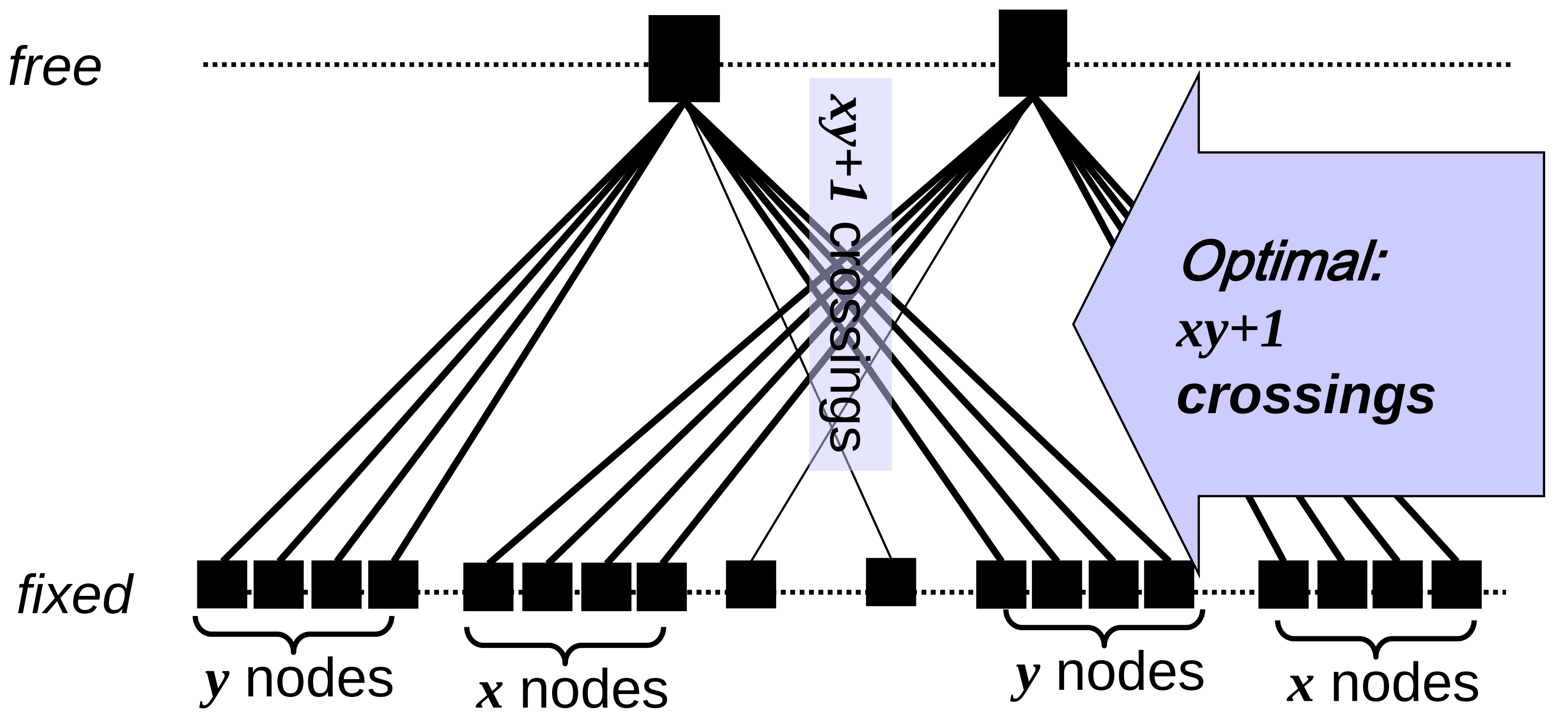
3. Median Method

- **Median placement:**



3. Median Method

- **Optimal placement:**



3. Median Method

- **Median: at most** $3xy$ **crossings**
- **Optimal: at least** $xy+1$ **crossings**

- *Theorem 2:* **The** *median method* **is at worst 3 times optimal.**

- **In practice, there are many good methods, and the median is just one of them.**

4. Integer Programming methods

- **Integer programming approach may be used for two-layer crossing problem**

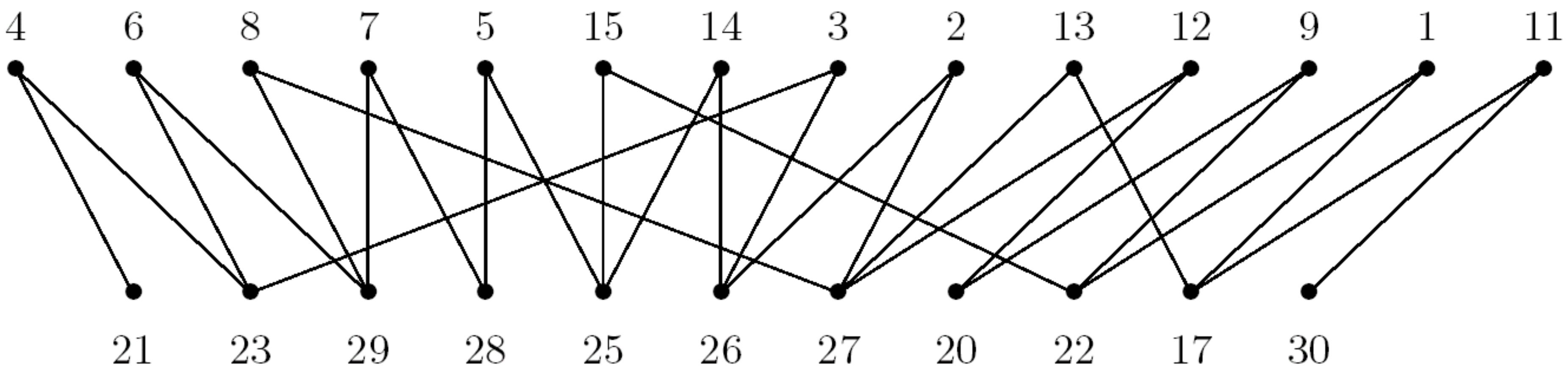
- **Solving integer programs require sophisticated technique: branch and cut approach can be used to obtain an optimal solution for digraphs of limited size [JM97]**

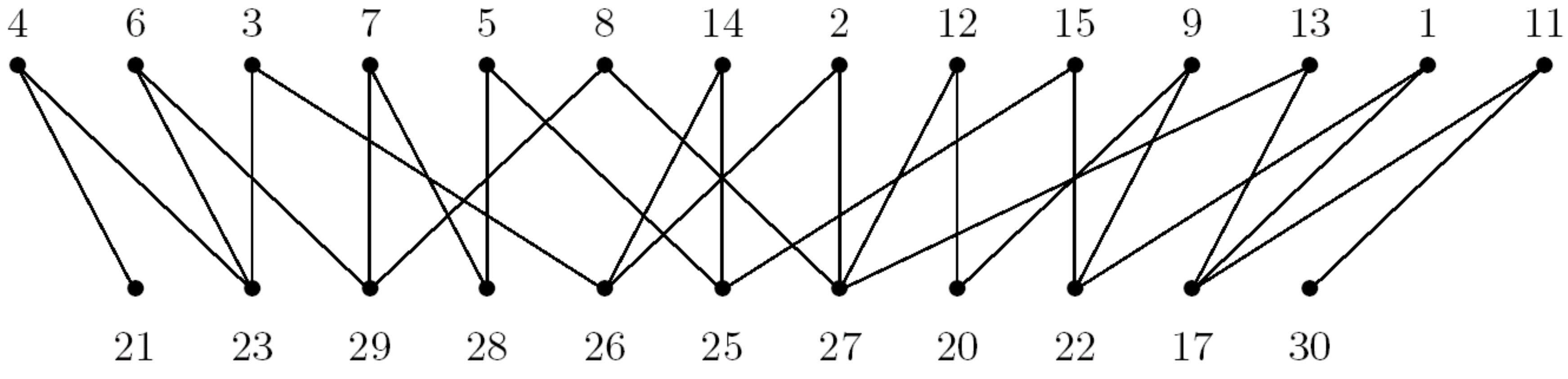
- **Advantage: find the optimal solution**
- **Disadvantage: no guarantee to terminate in polynomial time**

- **Successful for small to medium sized digraphs**

5. Planarization method [Mutzel97]

- **Use maximal planar subgraph approach**





Remark

- Median method seems very attractive

- Comparative tests
  - pseudo-random graphs [EK86, JM97]
  - real-world digraphs [GKNV93]
  - No single winner

- Use a hybrid approach
  1. Use the median method to determine the initial ordering
  2. Use an adjacent exchange method to refine

Step 4. Horizontal Coordinate Assignment

- Bends occur at the dummy vertices in the layering step.
- We want to reduce the angle of such bends by choosing an x-coordinate for each vertex, without changing the ordering in the crossing reduction step
- Optimization problem with constraints
  - draw each directed path as straight as possible
  - ensure the ordering in each layer (enforce minimal distance)
- It may affect the width of the drawing
- Some layered drawing requires exponential area with straight lines
- Quadratic programming problems can be solved by standard methods, but it requires considerable computational resource

Priority Barycenter Method

- Position vertices at the Barycenter of their neighbours
  – Can reuse “positions” from the ordering step
- Assign each vertex a priority
  – Priority = degree
  – Dummy vertices have the highest priority
- Enforce minimal distance between adjacent vertices
  – If two vertices are too close then move ONE of them to a safe distance
  – Move the vertex with the lower priority